MASTER IN COMPUTER SCIENCE

Using application level overlays to provide multicast services

Donnet, Benoît

Award date:
2003

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Using Application Level Overlays To Provide Multicast Services

Benoit DONNET

Promoteur : Professeur Olivier Bonaventure

Travail de fin d'études réalisé en vue de l'obtention du titre de Maître en Informatique

Année Académique 2002-2003

Abstract

These last years, the expansion of the multi-point communications, also called multicast, is impressive. Unfortunately, the multicast is difficult to implement efficiently because of scalability reasons, among others. A solution proposed is Application Level Multicast that could be implemented thanks to overlays. This dissertation is dedicated to the overlays. In a first time, we will give an overview of several overlays and we will compare them: Application Level Multicast Infrastructure, Narada, Overcast, Pastry, Yoid, Scribe and Reliable Overlay Network. Next, we will introduce the both overlays we implemented in Lancaster: Application Level Clustering and Tree Building Control Protocol. Their functioning and their limitations will be explained and a comparison between all the exposed overlays will also be proposed. Our implementation of Application Level Clustering and Tree Building Control Protocol will be presented by their design. The description of the various packages, classes and interfaces will constitute an introduction to the Java code. Finally, the assessment of the both mechanisms will be present by the discussion about the performance measures performed on the Planet Lab network.

Résumé

Ces dernières années, l'essor des communications multi-points, aussi appellées multicast, est impressionnant. Malheureusement, le multicast est difficile à implémenter efficacement en raison, notamment, de problèmes de mise à l'échelle. Une solution proposée est Application Level Multicast, qui peut être implémenté grâce aux overlays. Ce mémoire leur est consacré. Dans un premier temps, nous passerons en revue et nous comparerons différents overlays: Application Level Multicast Infrastructure, Narada, Overcast, Pastry, Yoid, Scribe et Reliable Overlay Network. Ensuite, nous présenterons les deux overlays que nous avons implémenté à Lancaster: Application Level Clustering et Tree Building Control Protocol. Leur fonctionnement et leur limitations seront expliqués et une comparaison entre tous les overlays exposés sera aussi proposée. Notre implémentation d'Application Level Clustering et Tree Building Control Protocol sera présentée au travers de leur design. La description des différents packages, classes et interfaces constituera une introduction au code Java. Enfin, l'évaluation de ces deux mécanismes sera présentée au travers de la discussion des mesures de performance effectuées sur le réseau Planet Lab.

Acknowledgements

I would like to acknowledge first my supervisor, Professor Olivier Bonaventure, and Cristel Pelsser for their patience towards me, for their advices (always wise), for their comments (always relevant) and their rereadings.

I would also like to thank all the members of the team of the Computing Department in Lancaster University for its welcome and its joviality. In particular, I would like to acknowledge three people: Doctor Laurent Mathy who accepted my training in Lancaster and who spent time to answer my questions. Doctor Steven Simpson who was always there to fill in my gaps and raise my misunderstandings. Finally, Mrs. Carol Airey who has dealt with the majority of my administrative steps.

Merci beaucoup à toutes et tous...

Contents

List of Figures

List of Tables

List of Pseudo-Code

Introduction

Since the beginning of the 90's, the Internet has seen a rapid expansion. Nowadays, a new kind of services arises. These services need to send data from one sender (the *source*) to a large number of receivers (called a *group of receivers*). These services are used by applications like games with players at different locations, video-conferencing, film streaming, TV streaming, ... The naïve way to implement this kind of services is to send the same data to each receiver. So, if the group of receiver is composed of 500 members, the source has to send 500 times the same data. The problem of this solution is obvious: it the resources inside the network.

An elegant solution was discovered by S. Deering [1, 2] to avoid this naive answer. It consists in building a logical distribution tree between the source and the receivers. The data is transmitted along the tree in only one data flow. This kind of communication is called multi-point communication or *multicast*.

However, the protocols used to implement multicast (PIM-DM, PIM-SM, ...) have shown their limitations. Because of the resources needed in routers, they are not scalable and thus, not adapted to large groups of receivers. These protocols are also difficult to implement, consume bandwidth and are expensive.

Nevertheless, the Internet Community didn't give up the multicast dream. An attractive alternative has been proposed: *Application Level Multicast* (ALM). ALM is a mean of enabling service provision in non-native multicast environments. ALM enables multicast-style communications to be conducted using only unicast messaging between parties in an ALM spanning tree [3].

ALM relies on protocols to build a tree between all participants of an ALM session. To achieve that, we use *Application Level Overlays* (overlays). Application level overlay networks consist of groups of application programs collaborating across a network using its basic, unicast services. This not only allows services such as multicast to be emulated, but also the participants can arrange themselves into trees or graphs according to application-specific criteria [4].

ALM, via overlays, allows to avoid some of the drawbacks of performing multicast at layer three: it is more scalable, it allows more flexibility, ... [5]

This dissertation treats overlays, and in particular the two overlays we implemented at Lancaster University: Application Level Clustering [6] and Tree Building Control Protocol [7].

This document is divided in three parts and is organized as follows: the first part (*Internet Protocol and Multicast*) contents two chapters. Chapter 1 introduces Internet Protocol (IP). We present the Internet Protocol Version 4 (Ipv4) and we discuss the need for a new version. The Internet Protocol Version 6 (IPv6) is then presented. The issue of the Ipv4-IPv6 transition are shortly introduced. The solution to the IPv4 limitations is also introduced and the IPv6 drawbacks are presented.

Chapter 2 is dedicated to multicast. We present the motivations of multicast communications. We also give an overview of protocols used to implement multicast at layer three: PIM-SM, PIM-DM, SSM, DVMRP, MBGP, MSDP and MOSPF. Finally, the drawbacks of these protocols are shown.

The second part (*Application Level Multicast and Overlays*) contents two chapters. Chapter 3 presents a solution to the drawbacks of multicast at layer three: Application Level Multicast. ALM is discussed by the way to implement it: Application Level Overlays. We will discuss several well known overlays: ALMI, Overcast, Narada, Pastry, Yoid, Scribe and RON. A comparison of the different overlays concludes this third chapter.

Chapter 4 presents in depth the two overlays we implemented in Lancaster: Application Level Clustering (ALC) and Tree Building Control Protocol (TBCP). The *distance* notion is first explained. Next, we present the functioning of both overlays. A simplified version of the finite state machine is also proposed. For space and easiness reasons, our fully specification of both overlays, including ABNF, messages syntax and semantic node behavior and a complete finite state machine, is proposed in appendixes A and B. The Maintenance Procedure and the heartbeat timer negotiation are presented. A comparison between the both overlays and their limitations are discussed. Finally, the comparison that concludes chapter 3 is extended to ALC and TBCP.

The third part (*Evaluation*) contents two chapters. Chapter 5 introduces our Java implementation of ALC and TBCP. The general structure of each overlay is presented. A description of the various packages, classes and interfaces is proposed to introduce our Java code. A CD-ROM is given in appendix. This CD-ROM includes the JavaDoc and the Java code of ALC and TBCP. This CD-ROM contains also an HTLM page (*readme.html*) to introduce the CD-ROM content. A section in chapter 5 explains the main problems we had to solve during the Java implementation. Finally, we discuss the way we could transfer data over the tree build by ALC and TBCP.

Chapter 6 discusses performance measures realized on the Planet Lab network. First, we present the Planet Lab network. Next, we present the results of the measures performed on ALC. The measure scenario is explained and the following points are discussed: construction of the tree, number of messages exchanged, heartbeat messages exchanged, `OBJREQ` messages exchanged, total bytes exchanged, tree maintenance and time needed to find a place in the hierarchy. The purpose of this discussion is to show that our implementation works and to measure the cost of the maintenance. Next, we introduce the measures performed on TBCP. These measures show the limitations of the Maintenance Procedure in TBCP. They also underline the fact that the time needed to find a place in the tree could be very high.

Part I

Internet Protocol and Multicast

Chapter 1

Internet Protocol

This chapter introduces the concepts of the Internet Protocol. First, we talk about IPv4 addresses in section 1.1. We give an overview of IPv4 and its limitations. For IPv6, in section 1.2, we introduce the changes compared to IPv4 and the solutions brought to the IPv4 problems.

1.1 IPv4

1.1.1 Overview

The Internet Protocol was defined in 1981 in [8]. In this section, we talk about the IPv4 addresses. Nowadays, IP version 4 is the most used version of the protocol.

IPv4 uses 32-bit addresses. The textual representation of such addresses is the following: d.d.d.d where each *d* is a decimal number interpreted as a byte of data. The first *d* represents the four most significant bits of the address and the last *d* the four less significant bits of the address. The fields are separated by ".". Here are some examples of IPv4 addresses: 138.48.160.121, 217.136.143.61, ...

Each IPv4 address is divided into two parts: the network part, used to identify the subnet in which the machine is and the local address part, used to identify the machine in the subnet.

It results from this division different forms of addresses, also called classes of addresses. The *Class A* address is indicated by the most significant bit of the address, which is always set to 0. As shown in figure 1.1.a, this class has a 7-bit network number and a 24-bit local address. The textual form of this address is *128.0.0.0/8* or *128.0.0.0 255.0.0.0* where "/8"[1] is call the subnet mask and indicates the network part length. This kind of address allows 128 class A networks. The *class B* address (figure 1.1.b) is indicated by the two most significant bits of the address, which are always set to 10. This class has a 14-bit network number and a 16-bit local address (for example: 138.48.0.0/16). The *Class C* address (figure 1.1.c) is indicated by the three most significant bits of the address, which are always set to 110. This class has a 21-bit network number and a 8-bit local address (for example: 138.48.160.0/8). The *Class E* address (figure 1.1.e)is indicated by the four most significant bits of the address, which are always sett to 1111. It represents addresses reserved for experimental use.

The *Class D* (figure 1.1.d)address is indicated by the four most significant bits of the address, which are always set to 1110. A class D address represents a multicast address.

[1]This notation is also known as the Classless Inter Domain Routing (CIDR) notation.

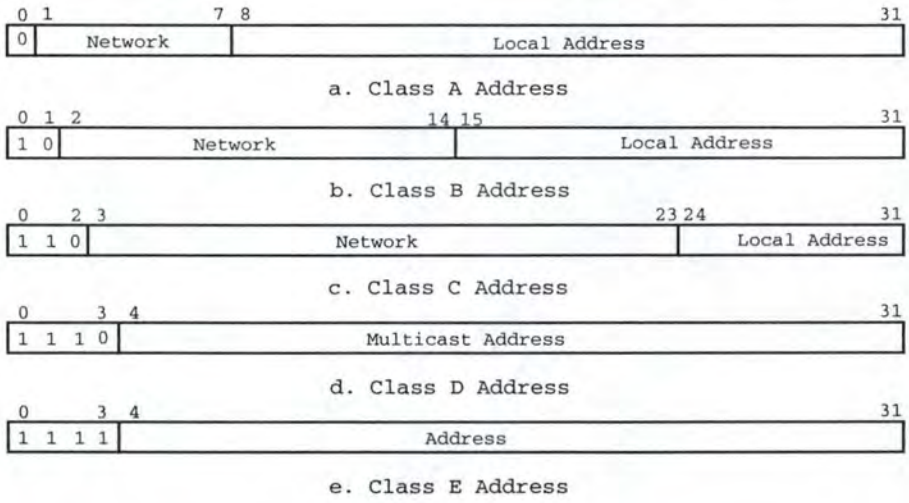


Figure 1.1: Address Class

The range of multicast addresses is thus 224.0.0.0 - 239.255.255.255. Special reserved group addresses (224.0.0.0 - 224.0.0.255) are defined. For example:

224.0.0.1 indicates all systems in this subnet.

224.0.0.2 indicates all routers in this subnet.

This division is, of course, theoretical. Now, for scaling reasons, we can find addresses with CIDR notation such as /23, /13, ...

IPv4 also defines reserved addresses. *127.0.0.1* is the local address of a machine. It is usually known as the "loopback address". *10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16* are used in private networks. *255.255.255.255* is used for general broadcast, i.e. sending of a packet to every local node. *224.0.0.0/8* to *239.0.0.0/8* represent the multicast addresses. *218.0.0.0/8* to *223.0.0.0/8*, *240.0.0.0/8* to *255.0.0.0/8* are reserved for a further use.

1.1.2 Limitations

There are several limitations to the IPv4 addresses. The chaotic address distribution leads to a squandering. For instance, the FUNDP has a /16 network (138.48.0.0/16). The University thus owns 2^{16} different addresses and that's too much for its needs! For this reason experts have introduced a less strict division by allowing the attribution of various subnet mask length.

There are geographical inequalities. Asia represents a strong growth potential but at the end of 2001, 74% of addresses were allocated to USA, 17% to Europe and 9% to Asia [9].

The growth of devices requiring Internet connections and permanent addresses (mobile services such as GPRS, UMTS, high bandwidth access, domotic applications, ...) is braked by the lack of addresses.

IPv4 is not developped for a commercial usage of Internet. IPv4 was not initially designed to support Quality of Services (Qos) functions, multicast, autoconfiguration, security (essential aspect of the commercial nowadays Internet).

The increasing size of routing tables due to solutions developed to tackle the lack of addresses (NAT, ...) leads to more complexity. In fact, this problems does not disrupt the BGP routing table but has rather a local influence. The Network Address Translator (NAT) is used to save IP addresses. The principle is simple: use private addresses in a small network (for example, inside a small company) and translate dynamically the packets sent/received. This solution generates problems with fragmented packets and protocols that encode addresses in the content of packets, such as FTP.

The mobility is not supported in a native way by IPv4. Incremental layers have been developed, such as Mobile IPv4, but they are optional and not really optimal.

QoS was not forecasted in the beginning and, nowadays, it is supported by using policies and tools created in addition of IPv4 (DiffServ, MPLS, Integrated Services, ...).

1.2 IPv6

IPv6 is a new version of the Internet Protocol and is aimed to replace IPv4. The changes from IPv4 to IPv6 are principally done to expand addressing capabilities (the IP address size is now 128 bits, instead of 32 bits). It also simplifies the packet format (header format simplification, improved support for extensions and options and flow labelling capabilities). It introduces autoconfiguration, multicast, routing optimization and mobile IPv6 in a native way. All these points are discussed in the further sections.

1.2.1 Expanded Addressing Capabilities

The IPv6 address is designed to identify an interface and a set of interfaces.

There are three ways to write the textual representation of an IPv6 address. x:x:x:x:x:x:x:x is the preferred form. Each x is an hexadecimal number representing a block of 16-bit of the address. Example:

fe80:0:0:0:201:2ff:fe29:85da

The example above shows an address containing blocks of bits set to zero. IPv6 authorizes to compress this address by using "::", which indicates multiple groups of 16-bit of zeros. The "::" can only appear once in an address. Here are some examples:

fe80:0:0:0:201:2ff:fe29:85da becomes fe80::201:2ff:fe29:85da

0:0:0:0:0:0:0:1[2] becomes ::1

0:0:0:0:0:0:0:0[3] becomes ::

::ffff:w.x.y.z is an IPv6 address representing an IPv4 address. It is useful for programs that are deployed on dual stack machines (i.e. machines that are IPv4 and IPv6 capable). They can use the same data structure and the same sockets. In the rest of the document, this kind of address is called an *Ipv4 Mapped Ipv6 Address.*

Ipv6 considers three kinds of addresses: unicast address, multicast address and anycast address.

[2]It is the loopback address

[3]The unspecified address

A *Unicast* address identifies a single interface. Not all unicast addresses are public. There are addresses that are guaranteed to be unique on a link (*link-local addresses*) and addresses that are guaranteed to be unique on a site (*site-local addresses*). The format of a unicast address is shown in figure 1.2. The text representation of IPv6 address prefix is the same than the CIDR notation for IPv4 prefixes. For example, 12ab:0:0:cd3::/60 represents a 60-bit prefix.

Figure 1.2: An IPv6 Unicast Address

A *Multicast* address identifies a set of interfaces. A packet sent to a multicast address is delivered to all interfaces corresponding to this address. Figure 1.3 shows the format of a multicast address. The eight most significant bits of the address (set to 1) identify the address as a multicast address. The *fg* field is a set of four flags. The high order three flags are reserved and have to be initialized to 0. If the fourth flag equals to 0, it indicates a "well-known" multicast address, i.e a permanently-assigned multicast address assigned by the global internet numbering authority (IANA). Otherwise, it indicates a "transient" multicast address, i.e. a non-permanently-assigned multicast address. The *sc* field is a 4-bit multicast scope value used to limit the scope of the multicast group, i.e. node-local scope, link-local scope, site-local scope and organization-local scope. The *group ID* identifies the multicast group (permanent or transient) within the given scope. For example, FF05::41:5cb9. The table 1.1 gives the explanation of the address.

Field	**Value**	**description**
fp	0xFF	Indicates that the address is a multicast address
Flags	0x0	Indicates that the address is a well-known address
Scope	0x5	Indicates that the scope of the multicast group is limited to the site
Reserved	0x0	Reserved for a further used. Always put to 0
Group ID	0x415cb9	Identifies the multicast group within the site-local scope. This value will be used at an Ethernet level to create a MAC address

Table 1.1: Explanation of an IPv6 address

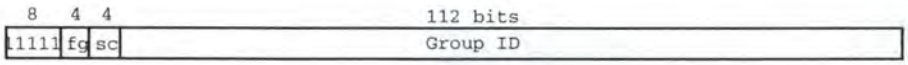


Figure 1.3: An IPv4 Multicast Address

An *Anycast* address identifies a set of interfaces. Compared to the multicast address, a packet sent to an anycast address is delivered to only one of the interfaces. As figure 1.4 shows

it, an anycast address is syntactically identical to an unicast address. The difference is the interface identifier is set to 0 in an anycast address.

Figure 1.4: An Anycast Address

1.2.2 Packet Format

1.2.2.1 Header Format

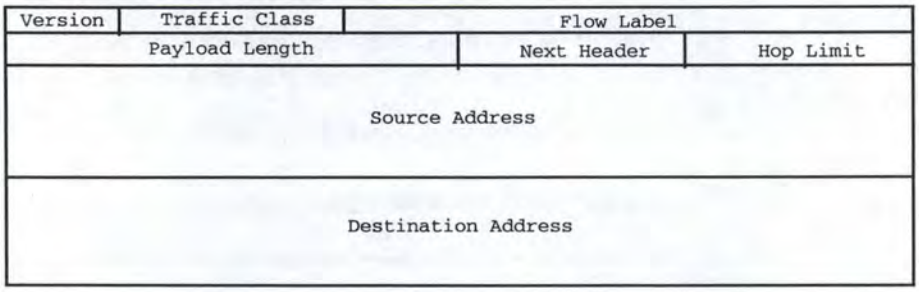


Figure 1.5: Header of an IPv6 packet

An IPv6 header contains several informations. The *Version* field indicates the Internet Protocol version number. The *Traffic Class* field is used for QoS. The *Flow Label* field is used by a source to label sequences of packets for which it requests special handling by the IPv6 routers, such as non-default QoS or "real-time" service. The *Payload Length* is an unsigned integer indicating the length of the IPv6 payload. Note that extension headers are part of the payload. The *Next Header* field identifies the type of header immediately following the IPv6 header. The *Hop Limit* field has the same purpose as the TTL (Time To Live) field in the IPv4 headers: indicate the maximum number of intermediate routers the packet can visit. The purpose if this field is to avoid the loop of packets inside the network. The *Source Address* and the *Destination Address* are 128-bit IPv6 addresses representing the sender and the receiver.

1.2.2.2 Improved Support For Extensions And Options

In IPv4, header options are encoded in separate headers that may be placed between the IPv4 header and the upper-layer header in a packet. An IPv6 packet may carry zero, one or more extension headers, each one identified by the Next Header field of the preceding header. Figure 1.6 shows examples of IPv6 header extensions.

Extension headers are not processed by any node along a packet's delivery path, until the packet reaches the receiver. However, there is an exception for the Hop-by-Hop Options header, which has to be examined by every node along a packet delivery path. For easiness reasons, this option is the first one encoded in an IPv6 packet.

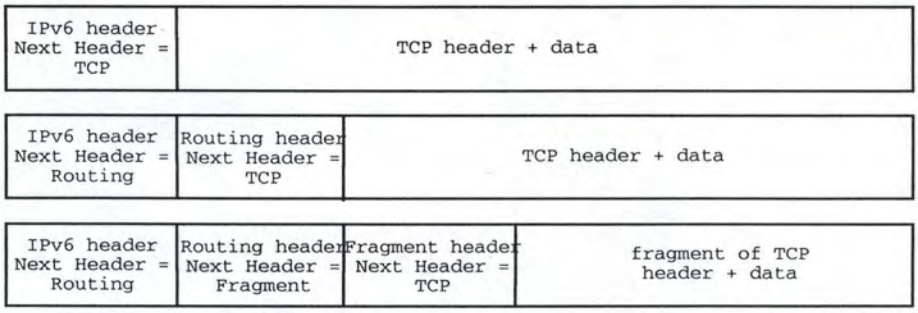


Figure 1.6: Examples of IPv6 header extensions

A full implementation of IPv6 includes the implementation of several extension headers. The *Hop-by-Hop Options* header is identified by a Next Header value of 0 in the IPv6 header and its format is shown by figure 1.7. The *Next Header* identifies the type of header imme-

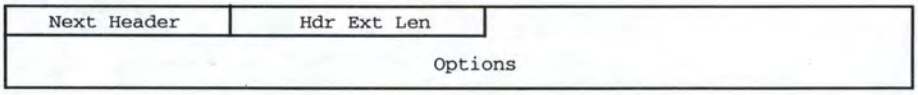


Figure 1.7: Hop-by-Hop extension header

diately following the Hop-by-Hop Options header. The *Hdr Ext Len* represents the length of the Hop-by-Hop Options header and the *Options* contains the options of the Hop-by-Hop Options header.

The *Routing* option is used by an IPv6 source to list one or more intermediate nodes to be absolutely visited on the way to the receiver. It is similar to IPv4's Loose Source and Record Route option. It is identified by a Next Header value of 43 in the immediately preceding header. Its format is shown by figure 1.8. The *Next Header* field identifies the type of header

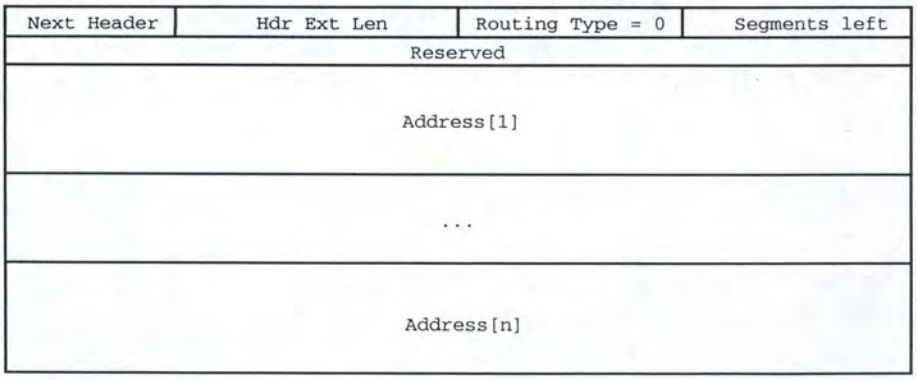


Figure 1.8: Routing extension header

immediately following the routing option header. The *Hdr Ext Len* represents the Routing

header length but not including the first eight octets. The *Routing Type* is always set to zero. The *Segments Left* field indicates the number of route segments remaining, i.e. the number of intermediate nodes still to be visited. *Reserved* is a reserved field and *Address[1..n]* is a vector of 128-bit addresses representing the intermediate nodes to be visited successively.

The *Fragment* option is used by an IPv6 source to send a packet larger than would fit in the path MTU (Maximum Transmission Unit) to its destination[4]. It is identified by a Next Header value of 44 in the immediately preceding header. Its format is shown by figure 1.9. The *Next Header* field identifies the initial header of the first fragment. The *Reserved* and

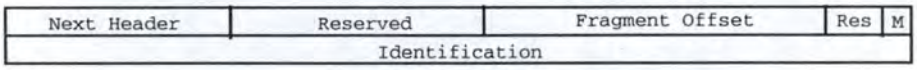


Figure 1.9: Fragment extension header

the *Res* fields are reserved, the *Fragment Offset* represents the offset of the data following this header. The *M* field is the more flag. If it equals 1, there are more fragments. If it equals 0, it is the last fragment. The *Identification* is a value generated by the source node for every packet that is to be fragmented and is used to recompose the packet at destination. It is also a packet identifier.

The *Destination Options* option is used to carry optional information(s) that need to be examined by the receiver(s). It is identified by a Next Header value of 60 and has the same format as figure 1.7.

The *Authentication* option provides a means to include optional authentication data. The inclusion of this authentication data allows the receiver to verify the authenticity of the packet sender, and also protects against modification of the packet. It may also be used to provide protection against replay of packets, such that saved copies of an authenticated packet can't later be resent by an attacker.

The *Encapsulating Security Payload* (ESP) [10] is used to provide, notably, confidentiality, data origin authentication and an anti-replay service. ESP may be applied alone, in combination with Authentication option or through the use of tunnel mode.

1.2.3 Autoconfiguration

IPv6 offers two types of autoconfiguration methods. The *Stateful Autoconfiguration* is the IPv6 equivalent to Dynamic Host Configuration Protocol (DHCP). DHCP's purpose is to enable individual computers on a IP network to extract their configurations from a server: the DHCP server. DHCPv6 is used to pass or addressing in the same way that DHCP is used in IPv4. It is called "stateful" because both DHCP server and client have to retain informations.

The *Stateless Autoconfiguration* [11] allows an host to propose an address (based on the network prefix identifying the subnet of the host given by a router and its MAC address) and proposes its use on the network.

1.2.4 Multicast

In Section 1.2.1, we talked about the format of an IPv6 multicast address.

[4]In IPv6, the fragmentation is performed only by the source node.

An IPv6 host must configure its network interface to accept packets sent with a given group destination address and must inform its local multicast router about its interest in receiving packets with a given group destination address.

IPv6 multicast uses different protocols depending on the equipment's location. On a LAN, the *Multicast Listener Discovery Protocol* (MLD) is used. It allows hosts to receive traffic from a specific set of sources and block the traffic from a specific set of sources.

Within the multicast domain, there is *Protocol Independent Multicast* (PIM) that can be declined in three different versions. The *Dense Mode* (DM) is similar to Distance Vector Multicast Routing Protocol (DVMRP developed in the next chapter) but can be used in combination with any unicast routing protocol. In the *Sparse Mode* (SM), the receivers are sparsely distributed and the shared trees are unidirectional. PIM-SM and PIM-DM for IPv4 are quite similar to their IPv4 versions that we will describe in the next chapter. The changes between the two versions are explained in [12]. With the *Source Specific Multicast* (SSM) [13, 14], multicast groups (*, G) are replaced by multicast channels (S, G). An IP packet is transmitted by a source S to an SSM destination address G, and receivers can receive this packet by subscribing to channel (S,G). The range FF3x::/96 of addresses is defined for SSM services.

In an inter-domain multicast, there is not any implemented protocol.

1.2.5 Routing Optimization

IPv6 allows a hierarchic addressing (by geographical area or by ISP), decreasing the number of routes for the backbone routers. It allows also to simplify the address aggregation by allowing to redistribute the addresses in an ideal way.

1.2.6 Mobile IPv6

The mobility support is an important aspect in IPv6 since the popularity of mobile computers (or assimilated devices, such as PDA, GSM, ...) doesn't stop to increase.

Mobile IPv6 [15] is intended to enable IPv6 nodes to move from one IP subnet to another. A *mobile node* is any node that may change its point of attachment from any IP subnet to another, while continuing to be addressed by its home address. A mobile node will be assigned at least three addresses. The *Home Address* is the permanent IP address assigned to a mobile node. It never changes, regardless of where the node is attached to the Internet. It is the identifier of a mobile IPv6 node. The *Care-of Address* is the mobile node's current address while away from home. It is a globally-routable address acquired by the mobile node through IPv6 address autoconfiguration (stateless or stateful) in the foreign subnet being visited by the mobile node. The *Link-local Address* belongs to a tiny part of the IPv6 addresses. It is not routable but it guarantees to be unique on a link, i.e. on a local network.

Mobile IPv6 considers two kinds of participants in a mobile session: the *Home Agent* and the *Correspondent Node*. The Home Agent is a router on the mobile node's home subnet maintaining a record of the current binding (i.e. the association between a mobile node's home address and its care-of address) of the mobile node. The Correspondent Node represents any node with which a mobile node is communicating.

Mobile IPv6 enables any IPv6 node to learn and store the care-of address associated with a mobile node's home address and then to send packets intended for the mobile node directly to it at this care-of address using an IPv6 routing header. Mobile IPv6 aims to avoid the

triangular routing by using a set of new IPv6 Destination Options. The *Binding Update* option sends by a mobile node to another to inform it of its current binding. The *Binding Acknowledgement* option sends by a node to acknowledge the receipt of a Binding Update. The packets carrying these options must use the Authentication option to avoid hijacking attacks, such as "remote redirection". Figure 1.10 shows the messages exchange.

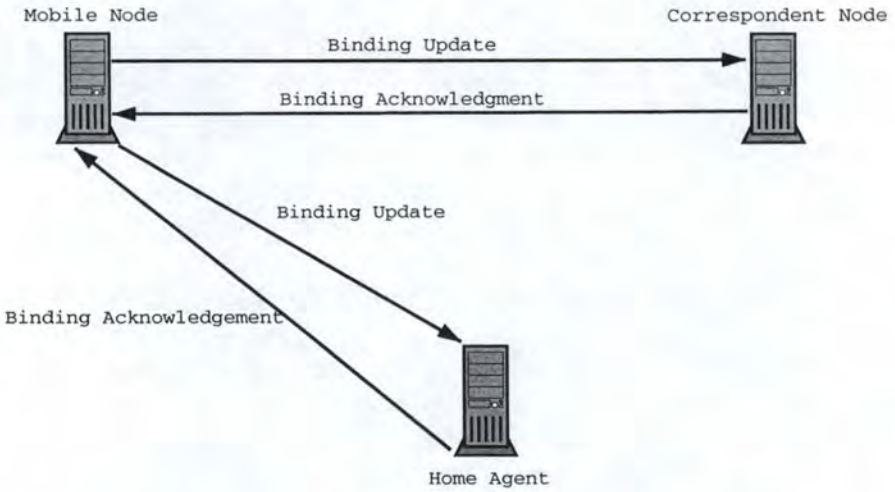


Figure 1.10: The Binding Messages

Two mechanisms have to be implemented to use Mobile IPv6. First, delivering packets to a mobile node from a correspondent node. Before sending a message, a node has to check its *Binding Cache* (a central data structure used to cache the mobile node bindings). If no entry is found for the destination, it sends the packet normally. The packet will be routed by usual routing mechanisms to the mobile node's home subnet. Then, the mobile node's Home Agent will intercept this packet and tunnel it to the care-of address. It is thus a case of traditional triangular routing. On the other hand, if the correspondent node has an entry for the destination in its Binding Cache, it will send the packet directly to the care-of address indicated by the binding, using an IPv6 Routing header. If a correspondent node receives persistent ICMP Host Unreachable or Network Unreachable messages after sending packets to a mobile node using its cached care-of address, it should delete the cache entry. Second, delivering packets to a mobile node from a Home Agent. When the Home Agent encapsulates a packet for delivery to the mobile node, the home agent uses the care-of address as destination address and uses its own address as source address. The home agent is expected to rarely send packets to the mobile node because the mobile node will send Binding Update messages as soon as possible to its correspondent node.

1.2.7 IPv4 - IPv6 Transition

An immediate migration from IPv4 to IPv6 is impossible for several reasons: cost, technology changes, economic failure, ... That's why the both protocols should co-exist during a while. Several approaches have been defined [16] to tackle this transition problem.

The key to a successful IPv6 transition is compatibility with the large installed base of IPv4 hosts and routers. The mechanisms the IPv6 routers and hosts may implement to be

compatible with IPv4 hosts and routers are: dual IP layer and IPv6 over IPv4 tunneling. There are two types of tunneling employed: *automatic* and *configured*. In the automatic tunneling, the IPv4 tunnel endpoint address is determined from the IPv4 address embedded in the IPv4 compatible destination address of the IPv6 packet. With the configured tunneling, the IPv4 tunnel endpoint is determined by configuration information on the encapsulating node.

The most easiest way for IPv6 nodes to remain compatible with IPv4-only node is by preserving a complete IPv4 implementation. These kind of nodes are called *dual-stack nodes* in this document. They are able to receive/send directly IPv4 and IPv6 packets. The dual stack technique may be used (but not necessary) in conjunction with the tunneling technique. In this case, the dual stack node can support configured tunneling or both configured and automatic tunneling. Thus, three configurations are conceivable: dual stack nodes that don't perform tunneling, dual stack nodes that perform configured tunneling and dual stack nodes that perform configured and automatic tunneling.

While the IPv6 infrastructure is being deployed, the existing IPv4 infrastructures can be used to carry IPv6 traffic. Tunneling provides a way to perform that. IPv6/IPv4 hosts and routers can tunnel IPv6 packets over areas of IPv4 only capable nodes by encapsulating them into IPv4 packets. This can be used in several ways. In the *Router-to-Router* way, the tunnel covers a segment of the end-to-end path that the IPv6 packet takes. In the *Host-to-Router* way, the tunnel covers the first segment of the end-to-end path that the IPv6 packet takes. In the *Host-to-Host*, the tunnel covers the entire path that the IPv6 packet takes. Finally, with the *Router-to-Host*, the tunnel covers the last segment of the end-to-end path that the IPv6 packet takes.

In the first two methods, the IPv6 packet is being tunneled to an intermediate router which has to decapsulate the IPv6 packet and then, forward it to the final destination. In this case, the endpoint of the tunnel is different from the packet destination. So, the address in the IPv6 packet being tunneled doesn't provide the IPv4 address of the tunnel endpoint. That's why we use the configured tunneling for this case.

In the last two tunneling methods, the IPv6 packet is tunneled to is final destination. The tunnel endpoint is thus the node to which the IPv6 packet is addressed. The tunnel endpoint can thus be determined from the destination IPv6 address of that packet (via the IPv4 Mapped IPv6 address). That's why we use the automatic tunneling in this case.

1.2.8 Solutions To The IPv4 Limitations

The adoption of the IPv6 addresses should allow to constitute a stock of 2^{128} unique addresses. It will be enough to tackle the growth of the Internet popularity for a while.

The adoption of IPv6 could reestablish the primal end-to-end system. It avoids the use of NAT that could cause problems with protocols that encode addresses in the message, such as FTP, Application Level Clustering (see chapter 4), Tree Building Control Protocol (see chapter 4), ...

IPv6 should allow a better management of multihomming, i.e. use several Internet Service Providers (ISPs) to provide Internet/Extranet services or Virtual Private Network (VPN[5]). This technic weights down strongly routing tables because the addressing becomes complex to link the different ISPs and the different parts of the firm networks.

[5] A Virtual Private Network is essentially a system that allows two or more private networks to be connected over a publically accessible network, such as Internet. It usually consists of an encrypted tunnel.

IPv6 owns several advantages allowing a better management [9] of QoS but they are not yet significant: the fixed header size, the header simplification, the flow label option, ...

1.2.9 Limitations

Some readers may be keen on IPv6 but it also has drawbacks. IPv6 could be a commercial failure. It is needed to convince ISPs to adopt the new version of the Internet Protocol and it will be not easy.

The transition IPv4 - IPv6 will cost a lot of money for ISPs to adapt applications, routers, ... This cost will be probably passed on clients, private persons or companies. We can note that the transition is perceived by the manufacturer as an economic leverage because of the need of material changes.

The backbone operators have to pass to IPv6. Otherwise, bottleneck problems can happen due to tunneling. An immediate conversion is impossible. The two protocols will have to coexist during a while. Solutions have already been developed for the coexistence (see section1.2.7).

An adaptation of all softwares (IPv4 to IPv6) is also needed. This adaptation could take time, even if each change is not really complex.

Chapter 2

Multicast

Multicast was already introduced in section 1.1.1 and 1.2.4 when we talked about addresses format and introduced routing protocols supporting multicast.

This chapter details multicast by presenting its motivations and problems. We also propose an overview of the protocols implementing multicast at layer three.

2.1 Motivations And Problems

The main motivation for multicast is to transmit the same information to several receivers. Thus, we can define multicast as the way to transmit a specific information from a single sender to a group of receivers. Examples: video conferencing, audio conferencing, video-on-demand, ... We note that, in some cases, the sender in a multicast session may change. The multicast session may have only one or more than one sender. Multicast was invented by S. Deering [1, 2]

There are two ways to implement multicast. The sender could send several times the same information, i.e. information is sent to each receiver separately or it could send only one copy of the information.

Figure 2.1 illustrates the first solution. *Member 1* has to send twice the same information. The drawback of this solution is the bandwidth consumption. *Member 1* uses bandwidth two times more than needed.

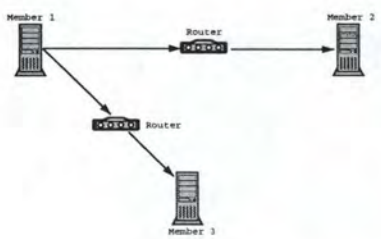


Figure 2.1: The first way to implement Multicast

In the second solution, the network will be responsible for forwarding efficiently this information to all receivers, as shown in figure 2.2. This solution is more efficient but, in a certain way, it is just a means to displace the problem.

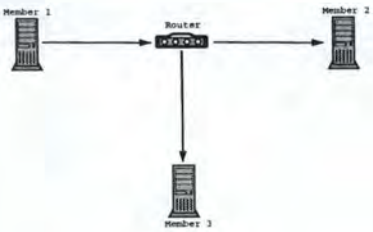


Figure 2.2: The second way to implement Multicast

Making the second solution available requires the development of protocols. Here are some protocols used to implement multicast at layer 3.

2.2 Overview

Protocol Independent Multicast Dense Mode (PIM DM) [17] assumes that when a source starts sending data, all downstream systems want to receive the information. Thus, the traffic is initially flooded to all PIM neighbors. If a branch does not have any group members, PIM DM will prune it off by setting up prune state, as shown in figure 2.3. This pruned branch timeouts after three minutes and traffic is re-flooded down the branch. The prune state contains source and group informations. The forwarding branches form a tree rooted at the source leading to all members of the group called *source rooted tree*. The mechanism used to broadcast datagram and prune unwanted branches is called the *reverse path forwarding* (RPF), a multicast forwarding mode where a data packet is accepted for forwarding if it is received on an interface used to reach the source in unicast. When a new member appears in a pruned branch, the branch can be grafted back. It avoids the new comer to wait three minutes before the next flooding. With PIM DM, only one source can send data. No rendez-vous point (i.e. a coordination point) is needed, contrary to PIM SM.

Protocol Independent Multicast Sparse Mode (PIM SM) [18] uses a *rendez-vous point* (RP) to coordinate forwarding from sources to receivers (see figure 2.4). Senders register to the RP via their first hop router. They will send single copy of data through the RP to the registered receivers. The receivers joined the *Shared Distribution Tree* rooted at the RP via their local *Designated Router* (DR). They will always receive data (and send messages such as Join[1] and Register[2]) through their DR. Each multi-access network has a designated router, which performs two main functions. First, it originates network link advertisements on behalf of the network and, second, it establishes adjacencies with all routers on the network. A Shared Distribution Tree is a tree whose root is a shared point (thus, it is the RP) in the network which multicast data flows down to reach the receivers in the network. The traffic is forwarded down the tree according to the Group address, regardless of source address. The notation used is (*, G), where "*" means any source and "G" is the group address. As this notation suggests, with PIM SM, several sources can send data.

[1] Join messages are coupled with Prune messages and are sent to join/prune a branch off of the the multicast distribution tree. This message disposes of a list containing a set of source addresses indicating the source-specific trees or shared tree that the router wants to join/prune

[2] Register messages are used by leaf routers attached to a source to register this source with the RP and to request the RP to build a tree back to these routers.

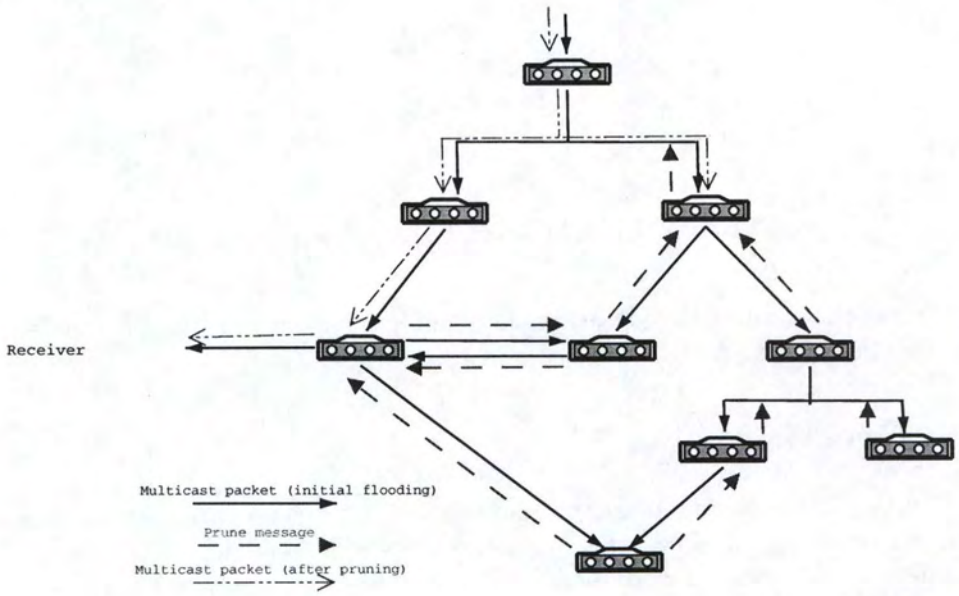


Figure 2.3: Pruning unwanted traffic

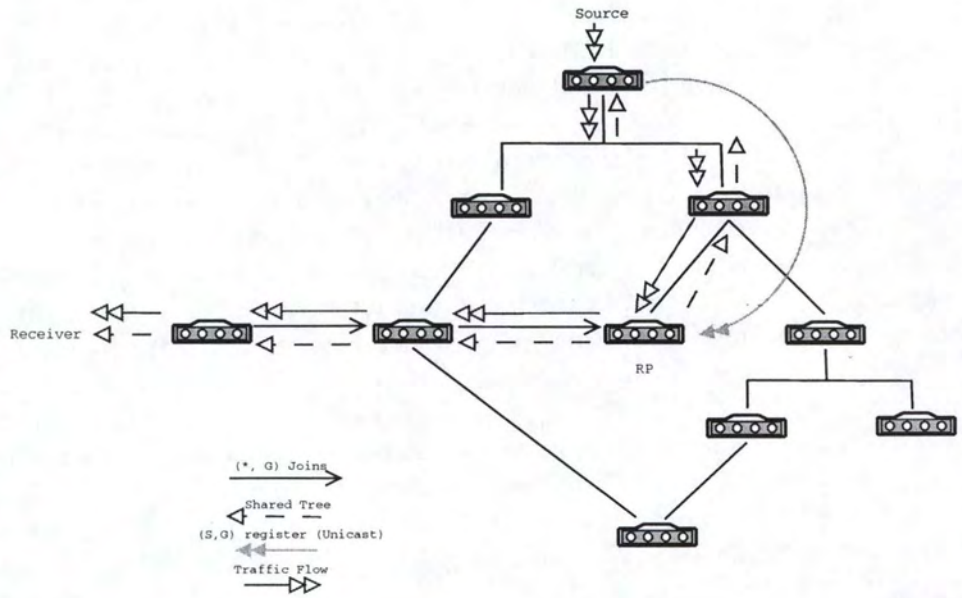


Figure 2.4: PIM SM register

PIM-SM and PIM-DM for IPv6 are quite similar to their IPv4 versions that we described above. The changes, essentially in messages format,between the two versions are explained in [12] and are not relevant to our work.

Source-Specific Multicast [13, 14] (SSM) is deployed by using PIM-SM. The network layer service provided by SSM is a *channel*, identified by an SSM destination IP address (G) and a source IP address S. An IPv4 address range (232.0.0.0 to 232.255.255.255) has been reserved by the IANA for use by the SSM service. For IPv6, the range FF3x::/96 is defined for SSM services. A source S transmits IP datagrams to an SSM destination address G. A receiver can receive these datagrams by subscribing to the channel (S,G). Channel subscription is supported by version 3 of the IGMP protocol for IPv4 and version 2 of Multicast Listener Discovery (MLD) protocol [19, 20] for IPv6. MLD is a subprotocol of ICMPv6. The purpose of MLD is to enable each IPv6 router to discover the presence of multicast listeners on its directly attached links and to determine specifically which multicast addresses are of interest to those nodes. An SSM receiver application must know both the SSM destination address G and the source address S before subscribing to a channel. Channel discovery is the responsibility of applications. This information can be made available in a number of ways, including via web pages, sessions announcement applications, ... SSM has several benefits. SSM lends itself to an elegant solution to the access control problem. When a receiver subscribes to an (S,G) channel, it receives data sent by a only the source S. SSM defines channels on a per-source basis, i.e., the channel (S1,G) is distinct from the channel (S2,G), where S1 and S2 are source addresses, and G is an SSM destination address. This averts the problem of global allocation of SSM destination addresses, and makes each source independently responsible for resolving address collisions for the various channels that it creates. SSM requires only source-based forwarding trees. This eliminates the need for a shared tree infrastructure, such as the RP-based shared tree infrastructure of PIM-SM. Finally, the SSM model is ideally suited for point-to-multipoint applications such as Internet TV.

Distance Vector Multicast Routing Protocol (DVMRP) DVMRP [21] is an "interior gateway protocol", suitable for use within an autonomous system (AS) but not between different AS. DVMRP (see figure 2.5) is a Distance Vector based routing protocol using some RIP[3] principles. Some fundamental differences with RIP are subnet masks that are sent in the route advertisements. In addition, DVMRP uses Poison-Reverse metrics and infinity. A Poison-Reverse metric is denoted by adding infinity[4] to the received metric and sending it back to the router from which it was received. It is used by DVMRP routers to signal their upstream neighbor that they are downstream and want to receive traffic from a source through their upstream neighbor. This is thus performed after the computing of the best path. DVMRP information is carried inside of Internet Group Management Protocol (IGMP) packets. DVMRP routes are used to build *Truncated Broadcast Trees* (TBT). A TBT, for a source subnet "S1", represents a shortest path spanning tree rooted at subnet "S1" to all other routers in the network. The multicast traffic is flooded down the distribution tree for a source and the downstream neighbors send back Prune messages for multicast traffic for which they have no group members.

Multiprotocol BGP (MBGP) [22] defines extensions for BGP to allow it to carry more information than just IPv4 route prefixes. MBGP does not propagate any multicast information nor builds any multicast distribution trees. MBGP can distribute unicast prefixes that can be

[3] *Routing Information Protocol.* It is a simple Distance Vector based routing protocol.

[4] different of mathematic infinity.

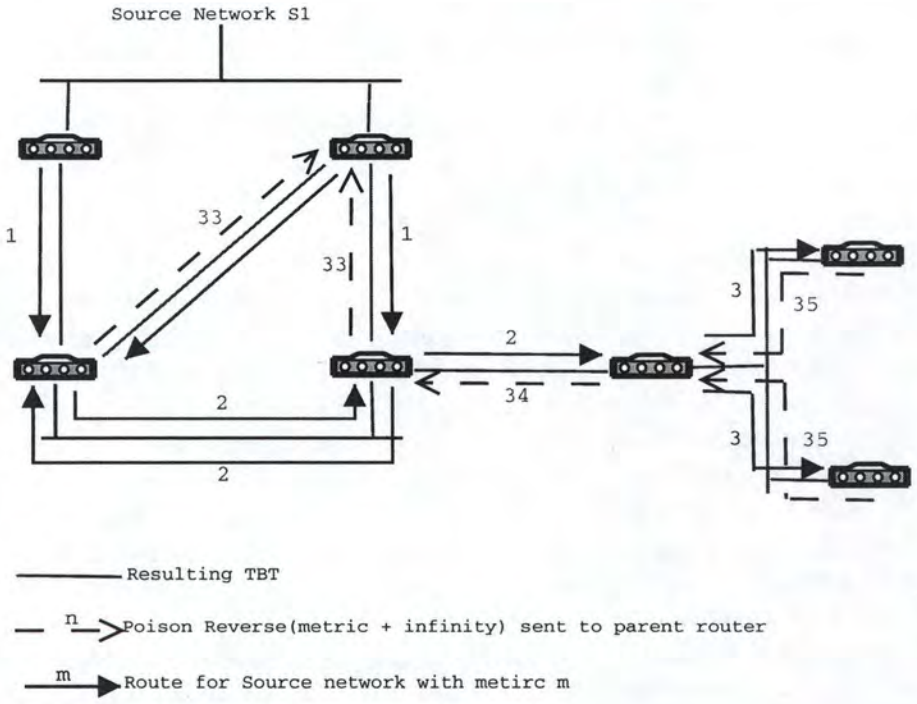


Figure 2.5: DVMRP - Source Tree

used for the multicast RPF check. The new types of routing information are: IPv4 prefixes for Unicast routing, IPv4 prefixes for Multicast and IPv6 prefixes for Unicast routing. This implies that MBGP can maintain several *Routing Information Bases* (RIBs): an Unicast RIB (U-RIB) and a Multicast RIB (M-RIB). The U-RIB contains the prefixes that were previously used by BGP. The M-RIB contains the prefixes used to RPF check for arriving multicast traffic.

Multicast Source Discovery Protocol (MSDP) [23] is a mechanism to connect multiple PIM SM domains together. MSDP routers in a PIM SM domain have a MSDP peering relationship with MSDP peers in another domain. This relationship is made up of a TCP connection in which control information is exchanged. When a RP in a PIM SM domain learns of a new sender, it sends a *Source Active* (SA) message to its MSDP peers. If the MSDP peer receives the SA from a non-RPF peer towards the originating RP, it will drop the message. Otherwise, it forwards the message to all its MSDP peers. When a MSDP peer which is also an RP for its own domain receives a new SA message, it determines if there are any group members within the domain interested in any group described by an (S,G) entry within the SA message. In this case, the RP triggers a (S, G) join event towards the data source. This sets up a branch of the source tree to this domain. Otherwise, it ignores the message. This procedure is called *flood-and-join.*

Multicast OSPF (MOSPF) [24] includes Multicast information in OSPF Link State Advertisements (LSA) to build multicast distribution trees. Each router maintains an image of the topology of the entire network. MOSPF uses a new type of LSA called *Group Membership LSA* to advertise the existence of group members. Group Membership are periodically flooded throughout an area in the same way as classical OSPF LSA. MOSPF uses Dijkstra algorithm

to compute shortest-path tree for every <source-network;group>.

All these protocols are complex, difficult to implement and have significant drawbacks: bandwidth consumption due to a lot of messages exchanges (for instance the flooding in PIM DM), state informations (routers have to retain informations for each group), cost (deployment of such protocols could be expensive).

Moreover, there are other drawbacks linked to multicast itself. Multicast is UDP[5] based, the support of high level functionalities (such as QoS) is difficult. UDP does not provide a reliable service. Packet drops may thus occur. There is no congestion control and it can result in network service degradation due to lack of TCP windowing and "slow start" mechanism. There are duplicate risks, i.e. some multicast protocol mechanisms can result in occasional packet duplication. There is a out-of-sequence packet problem because of network problems.

Nevertheless, the multicast based on protocols at layer three has some advantages. It enhances efficiency, i.e. the network bandwidth is used more efficiently since multiple streams of data are replaced by a single stream. It optimizes performance by eliminating traffic redundancy. It also allows the creation of distributed applications [25]. The multipoint applications will not be possible as demand and usage grows because unicast transmission will not scale. [25] explains this point like that: traffic level and clients increase at a 1:1 rate with unicast transmission but traffic level and clients do not increase at a greatly reduce rate with multicast transmission.

Meanwhile, the drawbacks created a need for a new model to implement Multicast that takes into account of all them. This alternative model is called *Application-Level Multicast* (ALM).

[5]User Datagram Protocol.

Part II

Application Level Multicast and Overlays

Chapter 3

Application Level Overlays

In this chapter, we talk about Application Level Multicast (ALM). First, we introduce ALM.

The second part of this chapter is dedicated to *Application Level Overlays*, a way to implement Application Level Multicast. In a first time, we give a definition of Application Level Overlays and, then, we present some Application Level Overlays. The two Application Level Overlays we implemented in Java at Lancaster University will be introduced in chapter 4.

3.1 Application Level Multicast

ALM is the implementation of multicast above the socket layer on end systems (hosts). All participants to an ALM session are connected with unicast links. A spanning tree is built between participants of an ALM session and is used to pass data between the nodes on the tree.

The source sends the information towards some receivers. Each receiver resends this information towards other receivers. As all is done with unicast, routers only see unicast connections. Figure 3.1 illustrates the concept. The participants to the ALM session are organized into a tree rooted at the source. The links between each node are simply unicast connections. When *Source* wants to transmit informations, it sends data to its children (*Receiver 1* and *receiver 2*) by using the unicast connections. If the receiver is a leaf (as Receiver 2), it does not forward data. If the receiver is a node (as Receiver 1), it forwards data to all its children by using unicast communications. And so on until the data has reached each receiver.

This method is more scalable because routers do not need to retain a lot of state information. All the group management is done at the application level and the participants are linked by unicast connections. The router has no particular role to play. ALM also allows more flexibility by customizing some tools such as error management, data transcoding, ... As ALM is developed at an application level, programmers can manage particular cases as they want, such as failure recovery, messages received in the wrong state, ... They can also develop their own way to encode/decode data for transmission to provide security, ... ALM offers also accelerated deployment, simplified configuration[1] and a better access control. The drawback is a higher traffic load since packets in ALM networks cannot be replicated at the exact branching points in the physical network and there are duplicated packets transmitted on some of the links [26].

[1]Now, it is the end user who is responsible for configuration/installation and no more the ISPs.

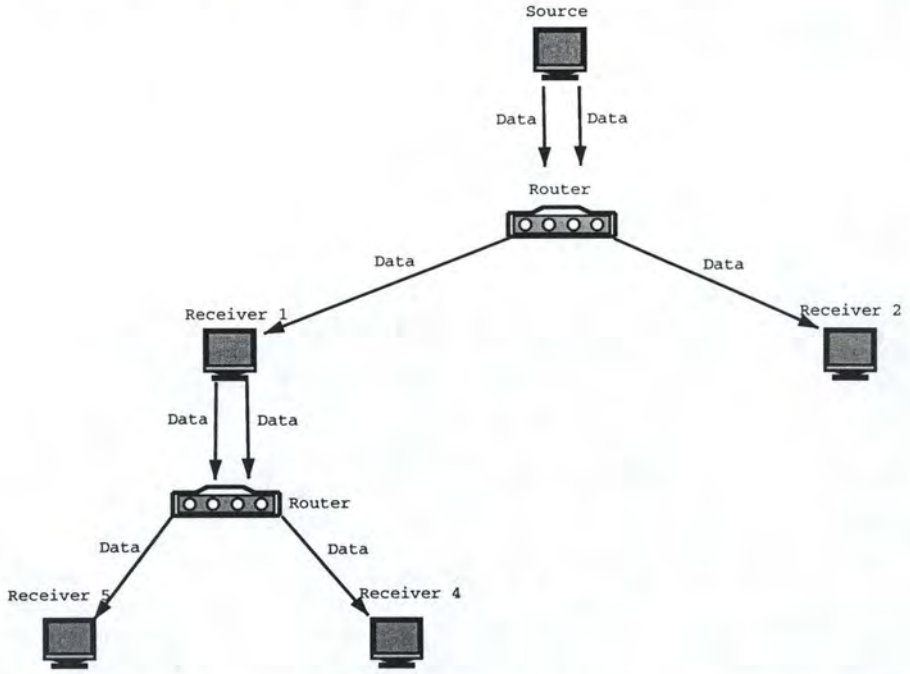


Figure 3.1: Application Level Multicast

3.1.1 Application Level Overlays: Definition

Application level overlay networks consist of groups of application programs collaborating across a network using its basic, unicast, services. This not only allows services such as multicast to be emulated, but also the participants can arrange themselves into trees or graphs according to application-specific criteria [4].

There are two different approaches to build an application level overlay [27, 28]. The first one builds the tree directly. Members explicitly select their parent from among the members they know. The second one creates a rich node connecting all the members. It is a centralized solution.

Application level overlay networks have some advantages [5]. It is incrementally deployable. An application level overlay network does not need any change in the existing Internet. It is adaptable. The set of links on which packets are sent is constantly optimized over metrics that are application-specific. By its adaptability and an increased control, an application level overlay network is more robust than multicast performed by the underlying layers. For example, with a sufficient number of nodes deployed, an overlay network may be able to guarantee that it is able to route between any two nodes in two different ways. It is customizable. The overlay nodes can be multi-purpose computers, easily outfitted with equipments needed. It is standardized. An application level overlay is built on the underlying network layers. This implies that overlay traffic will be treated as well as any other. For example, an overlay network can use TCP which is "simple", well known, network friendly (with regard to congestion control) and standardized.

Application level overlay networks have, of course, some drawbacks [5]. The management is complex. The manager of an overlay network is often physically far from the machines being

managed. Human interventions should be minimized and be possible by non expert people. In the real world, IP doesn't provide an universal connectivity service. Firewalls, Network Address Translator (NAT) and proxies are a source of problems. An application level overlay can't be as efficient as code running in routers. There is an information loss risk because an application-level overlay network is built above a network infrastructure that offers a nearly complete connectivity (modulo firewalls, NAT and proxies). We also expend effort deducing the underlying topology.

3.1.2 Overview

Several techniques for building overlay trees to emulate multicast services have been proposed. Here is an overview of some of them.

An *Application Level Multicast Infrastructure* (ALMI) [29] session consists on a *session controller* and several *session members*. A session controller is a program instance located at a site easily accessible by all the members. The session members are organized into a multicast tree where a line in this tree is represented by a unicast link between two members. This multicast tree is in fact a shared tree amongst the members. The purpose of the session controller is dual: ensure the connectivity of the multicast tree in case of departure/arrival of a member and in case of network failure, and ensure the efficiency by periodically computing a minimum spanning tree. This calculation is based on measurements made by members. A session member sends and receives data like in a classic multicast session. It also forwards data to designated neighbors and it monitors the performances of unicast links to and from other members. This monitoring is realized by sending probes and measuring the round-trip time (RTT). These measurements will be used by the session controller to compute the minimum spanning tree.

Figure 3.2 shows an example of ALMI tree. *SC* indicates the session controller. As shown in the figure, the controller is independent of the tree. In this example, D is a new comer. First, it has to contact the controller (its identity is known in advance). The controller will then assign randomly an existing member to the new comer. Each node periodically sends probes to its neighborhood and forwards the results to the controller. This information allows the controller to compute the shared tree and the results are communicated to all members in the form of a *(parent, children)* list.

Clearly, ALMI is a centralized structure. The session controller knows the complete topology and redistributes it to all members. To avoid problems linked to a centralized architecture, there are "back-up controllers" ready to relay in case of session controller failure. ALMI is based on TCP or UDP. The choice is dictated by its usage. A reliable service for data replication needs TCP. An unreliable service for stream-based application needs UDP.

ALMI is useful for multicast session where there is a great number of small groups, like video conferencing, multi-party network games, ... ALMI is also a solution for multi-sender multicast communications thanks to the centralized architecture.

Narada [28] targets principally groups of small size. It uses the first way to build an application level overlay, i.e. a new comer chooses itself its place in the tree. The tree building is done in two steps. First, Narada constructs a rich connected graph between the hosts, called *mesh*. Second, it constructs reverse shortest path spanning trees, each tree rooted at the corresponding source using well known distance vector routing protocols. For robustness reasons, each node has to conserve a complete information about the topology. This information must be updated each time a new member joins or an existing member

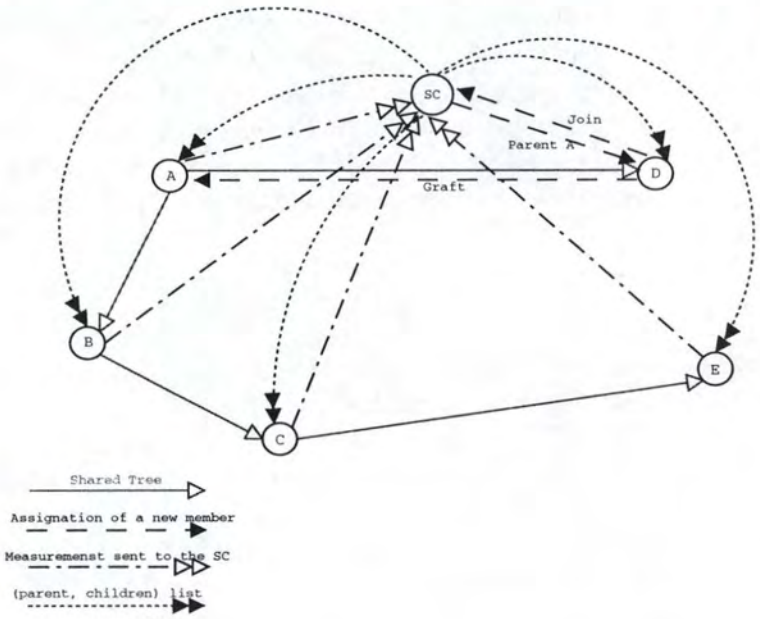
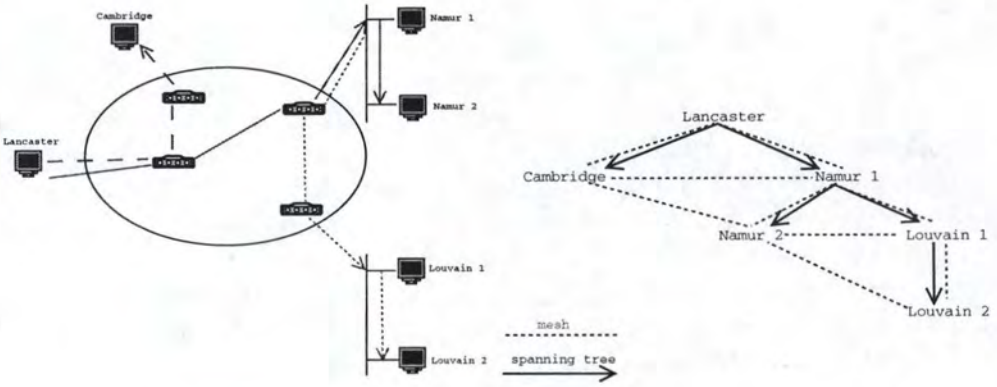


Figure 3.2: ALMI tree

leaves. As ALMI, Narada builds a multi-source tree.



(a) Network topology (b) Narada overlay tree

Figure 3.3: Narada tree

Figure 3.3 shows an example of Narada's tree building protocol[2]. Figure 3.3(a) gives the underlying network topology and figure 3.3(b) gives the overlay tree. As said above, Narada begins to build a rich connected graph (or mesh) between the hosts. This mesh is a graph where all nodes are connected together. The mesh on figure 3.3(b) is represented by the doted links. This mesh offers a richer topology. It allows robustness but it implies looping risks.

[2]For easiness reasons, the figure shows only a single source tree.

Based on this mesh, Narada builds the shortest spanning tree using well known distance vector routing protocols. This is indicated on figure 3.3(b) by the arrowed links. The main property of this spanning tree is a small delay from sources to receivers.

Overcast [5] aims to maximize the bandwidth of each node from the source and utilizes the underlying network topology efficiently. An Overcast system consists of a central source, internal Overcast nodes scattered in the network and standard HTTP clients located in the network. Overcast organizes the internal nodes into a distribution tree rooted at the source. The goal of Overcast's tree algorithm is to maximize bandwidth to the root for all nodes. The basic idea is to add a new node as far away from the root as possible but without compromising the bandwidth. The tree algorithm is run by the new comer. The new comer first contacts the root (acting as the current parent) and begins a series of rounds in which it will attempt to locate itself further away from the root without sacrificing bandwidth back to the root. A round consists on measuring bandwidth from root to new comer through each root's children. If the bandwidth through any of the children is as high as the direct bandwidth from the root, then a new round becomes with one of the children, acting as the new current parent. In case of multiple suitable children, the child closest in terms of network hops to the searching node is chosen. If no child is suitable, the search for a parent ends with the current parent. Using this distribution tree (see figure 3.4), Overcast provides Multicast group. Overcast allows also unmodified HTTP clients to join these multicast groups. A node periodically reevaluates its position in the tree by measuring bandwidth to its siblings, parent and grandparent. A protocol (*The Up/down Protocol*) is defined to maintain global state abóut the distribution tree.

The main drawback of Overcast is the root because it is a single point of failure. The solution brought is to replicate the root. Another problem is that the root maintains a complete information about the tree. This is solved by maintaining a tree in which first levels have degree one, as shown in figure 3.4. The advantage is that all nodes at these levels maintain complete full information about the tree. The drawback is a possible increase of delay.

Overcast is also a single-source multicast system and is based on TCP. Overcast may be used for data distribution, particulary for high resolution video.

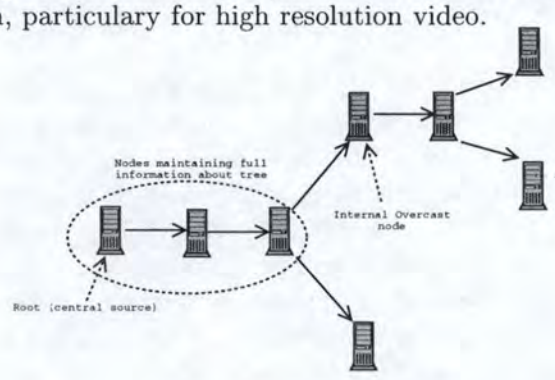


Figure 3.4: Overcast distribution tree

Pastry [30] aims to support a variety of Peer-to-Peer[3] applications, such as file sharing,

[3]A Peer-to-Peer (P2P) network is a distributed system in which all nodes have identical responsibilities and capacities and all communications are symmetric [30].

file storage, group communication and naming systems. A Pastry system is a self-organizing overlay network of nodes. Each node in the Pastry network has a unique, uniform random identifier (*nodeId*) in a circular 128-bit identifier space. When presented with a message and a numeric 128-bit key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes, as shown in figure 3.5. The expected number of forwarding steps in the Pastry overlay network is *O(log N)*, while the size of the routing table maintained in each Pastry node is only *O(log N)* in size (where N is the number of live Pastry nodes in the overlay network). Each Pastry node keeps track of its L immediate neighbors in the nodeId space (called the *leaf set*), and notifies applications of new node arrivals, node failures and node recoveries within the leaf set. Finally, Pastry takes into account locality (proximity) in the underlying Internet; it seeks to minimize the distance messages travel, according to a scalar proximity metric like the ping delay or the number of IP routing hops.

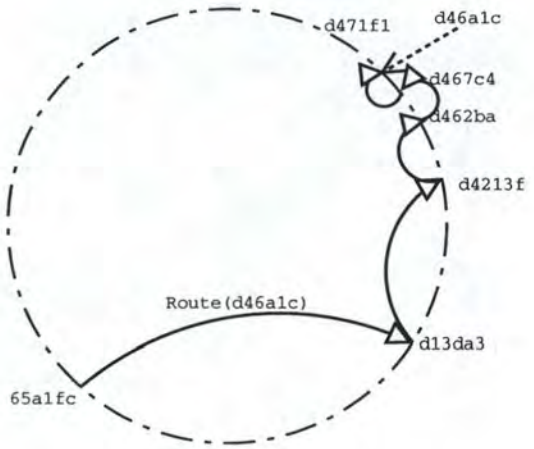


Figure 3.5: Routing a message from a node 65a1fc with key d46a1c. The dots depict live nodes in Pastry's circular namespace

Yoid [31] was previously called Yallcast. It is a suite of protocols that allows all of the replication and forwarding required for a given application to be done in the end hosts that are running the application itself. In other words, yoid works in the case were the only distributors of content are the consumers of the content themselves. The core of Yoid is a topology management protocol, called YTMP, that allows a group of hosts to dynamically auto-configure into two topologies. The first topology is a (tunneled) shared tree topology for efficient multicast distribution of application content. The second one is a (tunneled) mesh topology for robust broadcast distribution. This mesh topology is a node and its direct neighbors. The tunnel can be either two-party (i.e. based on TCP or UDP) or N-party (based on a very tightly scoped IP multicast). Each host can join or leave the two topologies independently, making the group itself dynamic. Each group has one or more rendez-vous points. A new comer in a group first contacts the rendez-vous point. The rendez-vous point answers by sending back a list of nodes and the prospective receiver chooses one of them as a parent. To leave the group, a member contacts the rendez-vous point.

Figure 3.6 shows a yoid tree and gives terminology associated with it. Each box represents a member. The rendez-vous point is not shown. The solid arrows represent the "links" of the

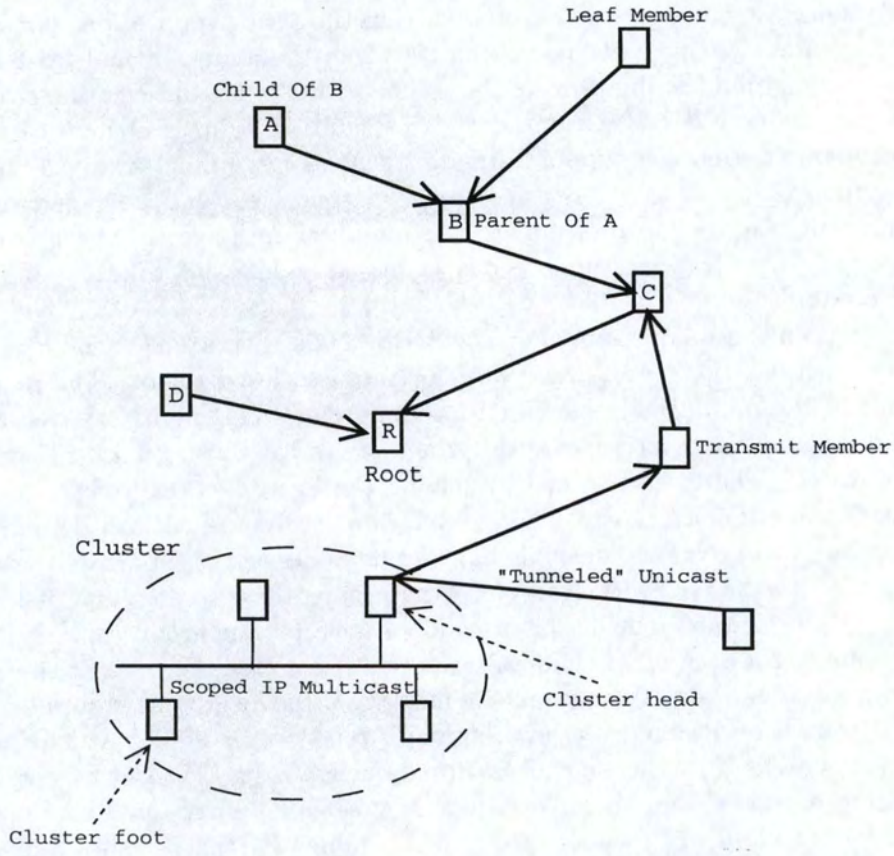


Figure 3.6: Yoid Tree

tree. These links refer to the relationship between neighbor members in the tree. A member may receive and transmit either via unicast IP or scoped IP multicast. The relationship between two neighbor members over unicast IP is that of parent/child. Where IP multicast is used, the set of members is grouped as a *cluster*. One member of the cluster is elected the head and is responsible to bridge the cluster via unicast IP with the rest of the tree. The other cluster members are called *feet* and transmit/receive data to/from the tree via the head. Each yoid tree must have a single root. Each member, at a given time, is a transmit member or a leaf member. It depends on whether it has a multiple neighbors or a single neighbor respectively.

Scribe [32] is a fully decentralized model and is built above Pastry. A Scribe system consists of a network of Pastry nodes, where each node runs the Scribe applications software.Any Scribe node may create a *group*; other nodes can then join the group, or multicast messages to all members of the group. Scribe provides best effort delivery of multicast messages, and specify no particular delivery order. Groups may have multiple sources of multicast messages and many members. Scribe can support simultaneously a large numbers of groups with a wide range of group sizes. It uses Pastry to manage group creation, group joining and to build per-group multicast tree to disseminate the multicast messages in the group. Pastry and Scribe are fully decentralized: all decisions are based on local information, and each node has identical capabilities.

Each group has a unique *groupId*. The Scribe node with a *nodeId* numerically closest to the groupId acts as the *rendez-vous point* for the associated group. The rendez-vous point is the root of the multicast tree created for the group. This multicast tree will be used to disseminate the multicast messages in the group. It is created using a scheme similar to reverse path forwarding. The tree is formed by joining the Pastry routes from each group member to the rendez-vous point. Group joining operations are managed in a decentralized manner to support large and dynamic membership. Scribe nodes that are part of a group's multicast tree are called *forwarders* with respect to the group; they may or may not be member of the group. Each forwarder maintains a *children table* for the group containing an entry (IP address, nodeId) for each of its children in the multicast tree. When a Scribe node wishes to join a group, it asks Pastry to route a `JOIN` message with the group's groupId as the key. This message is routed by Pastry towards the group's rendez-vous point. At each node along the route, Scribe checks if the new comer is already a forwarder. If so, it accepts the node as a child, adding it to the children table. If not, it creates an entry for the group and adds the source node as a child in the associated children table. It then becomes a forwarder for the group by sending a `JOIN` message to the next node along the route from the joining node to the rendez-vous point. Figure 3.7 illustrates the group joining mechanism. The circles represent nodes. We assume that there is a group with groupId 1100 whose rendez-vous point is the node with the same identifier. The node 0111 is joining the group. In this example, Pastry routes the `JOIN` message to node 1001; then, the message from 1001 is routed to 1011; finally, the message from 1011 arrives at 1100. This route is indicated by the solid arrows. Let us assume that nodes 1001 and 1101 are not already forwarders. The joining of node 0111 causes the other two nodes along the route to become forwarders for the group, and causes them to add the preceding node in the route to their children table. Now, if node 0100 decides to join the same group, the route that its `JOIN` message would take is shown using the dot-dash arrow. Since node 1001 is already a forwarder, it adds node 0100 to its children table for the group, and the message is terminated.

Periodically, each non-leaf node in the tree sends a heartbeat message to its children. A

child suspects that its parent is faulty when it fails to receive heartbeat messages. Upon detection of the failure, a node calls Pastry to route a `JOIN` message to group's identifier. Pastry will route the message, thus repairing the multicast tree. For example, figure 3.7 considers the failure of node 1101. Node 1001 detects the failure and uses Pastry to route a `JOIN` message towards the root through an alternative route (indicated by the dashed arrows). The message reaches 1111 that adds 1001 to its children list and forward the `JOIN` message towards the root. This causes node 1100 to add 1111 to its children list.

Finally, Scribe is based on TCP.

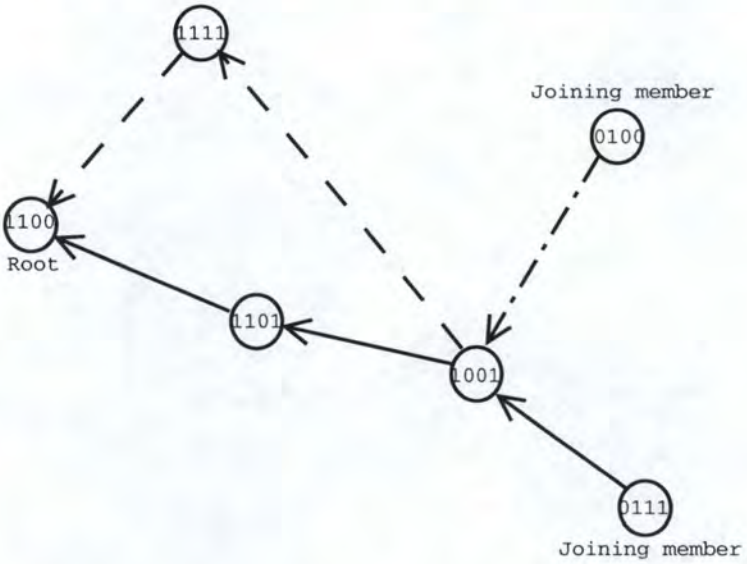


Figure 3.7: Membership management and multicast tree creation with Scribe

Resilient Overlay Network [33] (RON) is a wide-area network overlay system that can detect and recover from path outages and periods of degraded performance within several seconds. RON pursues three goals. The main goal of RON is to enable a group of nodes to communicate with each other in the face of problems with the underlying paths connecting them. RON detects problems by aggressively probing and monitoring the paths connecting its nodes. If the underlying Internet path is the best one, that path is used and no other RON node is involved in the forwarding path. If the Internet path is not the best one, RON will forward the packet by the way of other RON nodes. RON nodes exchange information about the quality of the paths among themselves via a routing protocol and build forwarding tables based on a variety of path metrics, including latency, packet loss rate, and available throughput. The second goal of RON is to integrate routing and path selection with distributed applications more tightly than is traditionally done. This integration includes the ability to consult application-specific metrics in selecting paths, and the ability to incorporate application-specific notions of what network conditions constitute a "fault". An example of this use is video-conferencing. This idea can be extended further to develop an *Overlay ISP*, formed by linking (via RON) points of presence in different traditional ISP's after buying bandwidth from them. Using RON's routing machinery, an Overlay ISP can provide more resilient and failure-resistant Internet service to its customer. The last goal is to provide a framework for the implementation of expressive routing policies, which govern the choice of

paths in the network. For example, RON facilitates classifying packets into categories that could implement notions of acceptable use, or enforce forwarding rate controls.

Figure 3.8 shows an example of the RON general approach. Nodes get measurements of some properties of the paths between them by sending probes. This is shown on the figure by the dotted links. The nodes then exchange this information with each other so that they can do some routing based upon this information. Once they have exchanged this information, they then route the data over this path if it is better than the underlying Internet. This is indicated on the figure by the dashed link.

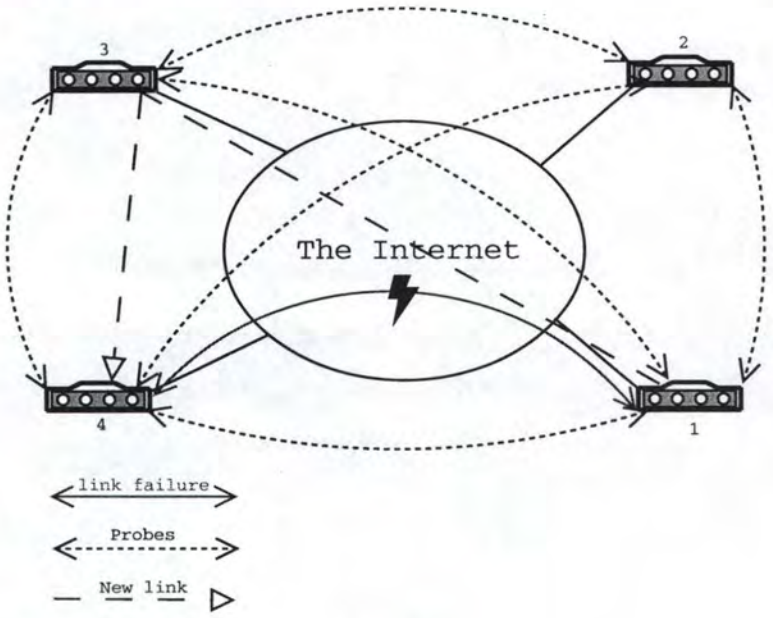


Figure 3.8: RON overlay network

As a summary of this overview, we propose the table 4.1. The *TCP* and *UDP* columns indicate if the overlay is based on TCP or UDP. The *centralized* column indicates if the overlay provides a centralized solution or not. The *Uni-Source* column indicates if the tree build by the overlay protocol is uni-source or not. The column *Full knowledge* indicates if, in the tree, there is at least one node that have a full knowledge of the topology. The *Interdomain* column indicates if the overlay is used for interdomain traffic or not. The *Tree* column gives an information about the type of tree build by the overlay protocol. The last column (*Used for*) indicates the utilization of the overlay by an application. The *?* indicates that the information is not given by the authors.

Overlay	TCP	UDP	Centralized	Uni-source	Full knowledge	Interdomain	Tree	Used for
ALMI	Y	Y	Y	N	Y	N	Shared tree	Multicast sessions where there is a great number of small groups
Narada	?	?	N	N	Y	N	Shortest path	Small and sparse group
Overcast	Y	N	N	Y	?	N	Distribution tree rooted at the source	On-demand and live data delivery
Pastry	?	?	N	N	N	N	Circular name space	P2P application
Yoid	Y	Y	N	Y	Y	N	Mesh topology and shared tree	Replication, data distribution
Scribe	Y	N	N	N	N	N	Multicast tree	Multicast
RON	N	Y	?	?	?	Y	?	Monitor the functioning and quality of the Internet paths

Table 3.1: Application-level overlays overview summary

Chapter 4

Two Application Level Overlay Protocols

In this chapter, we describe in details two Application Level Overlay protocols developed at Lancaster University: Application Level Clustering (ALC) and Tree Building Control Protocol (TBCP).

The purpose of these overlays is to build a logical tree between end-hosts. The tree should be optimal and should minimize the distance between the nodes. The Dijkstra algorithm cannot be used because a node has only a partial view of the tree building session participants. Furthermore, a node has no knowledge of the network topology. Once the tree is built, to make sure a node is at its best place and the tree is still optimal, a Maintenance Procedure has to be defined.

This chapter first introduce the notion of distance (section 4.1). Then, we describe Application Level Clustering (ALC) [6], an overlay protocol building a non-constrained tree (section 4.2). For easiness and space reasons, we will only describe how ALC works, i.e. builds a logical tree between the nodes. Our fully specification, including ABNF, messages syntax and semantic, we used to implement ALC in Java is presented in appendix A. Next, we develop Tree Building Control Protocol (TBCP) [7], an overlay protocol building a constrained tree (section 4.3). Again, for easiness and space reasons, we only describe how TBCP builds a logical tree between the nodes. Our fully specification, including ABNF, messages syntax and semantic is presented in appendix B. A comparison of both overlays is proposed in section 4.4. The Maintenance Procedure is also presented (section 4.5). A mechanism presenting the heartbeat timer negotiation is described (section 4.6). The limitations of both overlays are shown in section 4.7. Finally, a summary is given in section 4.8.

4.1 Distance

Both protocols include the notion of distance. [6, 7] let the designer free to decide the kind of distance he wants to implement. It could be *performance* metric (RTT, delay, bandwidth, ...), *semantic* metric (content, features, ...), or even a multi-metric distance space. The only requirement is that the measures are "comparable".

For both protocols, we chose a *performance* metric and, more precisely, a round-trip time (RTT).

We didn't use the Echo Request of ICMP and the port 7 (UDP Echo) because some

Pseudo-Code 1: Pseudo-code for the RTT measure

```
int i = 0;
long[] sendTime, receiveTime = new long[3];
long rtt = 0;

//The packet is used to measure the rtt.
Datagram packet = "HELLO";
while (i < 3)
   {
      sendTime[i] = currentTime;
      socket.send(packet);
      socket.receive(response);
      receiveTime[i] = currentTime;
      rtt += receiveTime[i] - sendTime[i];
      i++;
   }
rtt = rtt/3;
```

routers do not take account of Echo Request or manage them with a low priority. There is another potential problem with the firewalls. Some firewalls automatically reject the Echo Request.Thus, there is a risk of false indications.

Pseudo-code 1 shows the way we choose to implement the RTT measurement, and more precisely the way we send probes to evaluate the RTT. For relevant reasons, the pseudo-code 1 does not show the connection opening and closing but the port used is not a well-known port. The port choice is based on the one used by the protocols for messages exchange.

The RTT is given by the mean average of three successive ping. We chose three because we wanted an arbitration between the time needed to perform the measure and the measure accuracy. This accuracy was proved by the measurement performed on Planet Lab (see chapter 6).

The RTT measure has to be performed within one second, otherwise, the node starts again the measure.

Our solution for the measure has a drawback: the port used can lead to problems with firewalls or can be already used by another application.

In the rest of this document, when we are going to talk about distance, we will understand RTT.

4.2 Application Level Clustering (ALC)

This section describes the formation of application-level overlay networks by allowing its participants (*peers*) to organize themselves hierarchically in clusters [6]. A cluster is represented by a cluster head and is composed of the cluster head and other nodes. The hierarchy of clusters is organized into *layers*, where layer L_i is composed of the heads of clusters that divide L_{i-1}. For instance, in figure 4.1,the L_1-cluster headed by R is composed of A and B.

The cluster hierarchy built like this forms thus a logical tree spanning all the cluster heads and rooted at layer L_0 (see figure 4.1 right). The cluster heads could be seen as the nodes

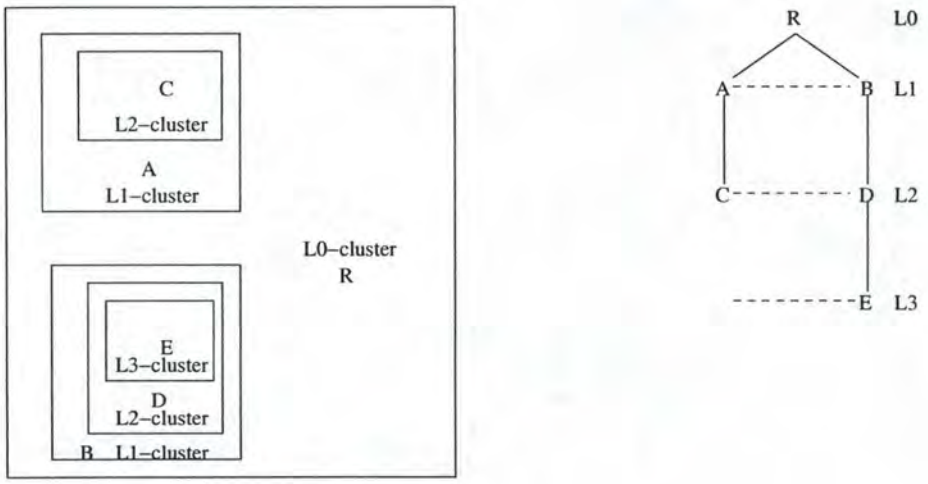


Figure 4.1: Cluster hierarchy

in an Application Level Overlay session. Each node at layer L_i has to record information only for its parent cluster head (the L_{i-1} cluster head) and the L_{i+1} cluster heads (also called children). For instance, in figure 4.1, B records R as its parent and D as its child.

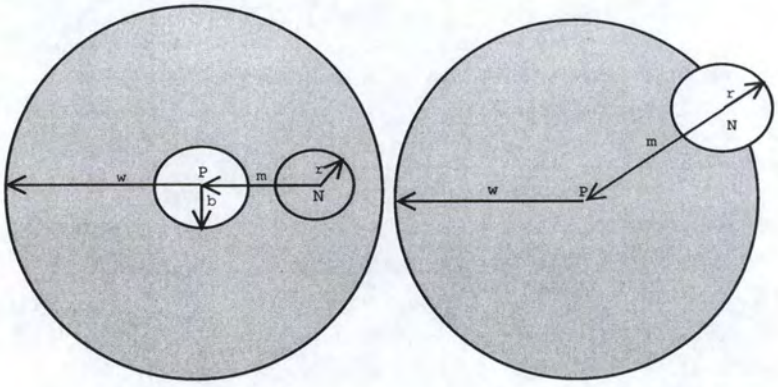
The algorithm used to build the hierarchy is distributed, recursive and based on unicast communications. It doesn't need any knowledge of the network topology neither any prerequisite knowledge of the full hierarchy. The only information needed about the hierarchy is the root address.

The Join Procedure is built like this: A node (N) desiring to join the hierarchy first measures its distance to the root of that tree, acting as potential parent, by sending probes and measuring the RTT. Then, it sends this *potential* parent (P) a `JOIN` message containing this distance. Based on this information, the potential parent computes the *zone* of the joining node. A zone is a ring centered on this node (the potential parent, thus P in figure 4.2). More precisely, a zone (or a region) is a triplet <best, worst, radius>, where *best* is the inclusive measurement for suitable children, *worst* is the exclusive measurement for suitable children and *radius* is the worst measurement that the prospective peer (i.e. the new comer, thus N in figure 4.2) should accept from the selected children. The radius is built by using the formula $10^{\lfloor log_{10}m \rfloor+1} - 10^{\lfloor log_{10}m \rfloor}$, where m represents the measure performed by the new comer (N)[1]. The original plan was not to have general formulae to categorize children but divide them according to fixed boundaries, and the measure will fall between two of them. $log_{10}m$ rounded down to an integer gives the number of the lower boundary, and the radius is set to the width between the two boundaries. If the new comer is a very remote peer talking to the root, the large radius will mean that even the best of the peers that the new comer selects as the next potential parent could be a long way away. As the new comer moves down the levels, the distance between it and the current potential parent decreases rapidly, and the new comer quickly refines its position in the hierarchy. So, initially, the new comer gets a very coarse set of parents to try (i.e. roughly in its area), but it reduces exponentially with each level.

Figure 4.2(a) shows an example of region, where w means the worst, b the best, r the radius and m the measure performed by the new comer (N). The resulting region is in grey.

[1]The formula was suggested by Steven Simpson from the Computing Department at Lancaster University.

For the protocol implementation, we chose to build the region by allocating zero to the best and the measure to the worst. It results a region, as the one shown in figure 4.2(b).



(a) A region (b) A region implemented

Figure 4.2: A region

At this moment in the join procedure, Two cases are possible:

1. The joining node is the only node in this zone.
2. There are other nodes in the zone.

In the first case, N becomes a child of P (see figure 4.3). The parent acknowledges the joining node by sending a `NCA` (NEW_CLUSTER_ACK) message. This message indicates to N that it has found its place in the hierarchy.

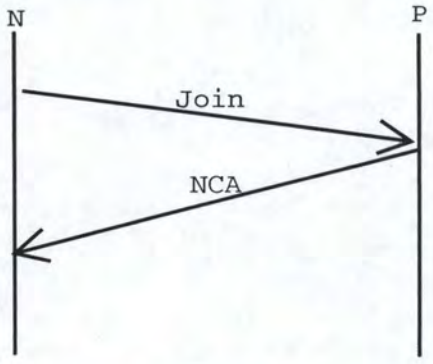


Figure 4.3: Application Level Clustering - Join Procedure (first case)

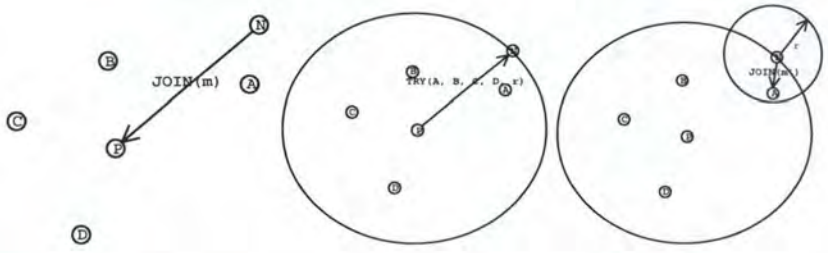
The second case is a little more difficult. The potential parent sends back a `TRY` message containing the list of all the nodes in the zone of the joining node. This message also contains the radius information, used by N to build a region. N then measures its distance to some of the nodes in the list. To ensure the scalability of the protocol and to control the latency

of the Join Procedure, each node disposes of a maximum limit L. It represents the number of nodes N considers at each step of the algorithm. If the number of nodes in the list is smaller than L, Peer 1 measures its distance to all of them. Otherwise, it randomly picks L of them. This procedure is called *Probabilistic Join*. Again, two cases are considered:

1. At least one node falls within the joining node's scope, i.e. is in the region built with the radius as in figure 4.4(c).

2. All the nodes considered are outside the joining node's scope.

Figure 4.4 illustrates the first case. Figure 4.4(a) illustrates the first step of the Join Procedure: the new comer (N) considers the root of the hierarchy (P) as potential parent and contacts it by sending a `JOIN` message containing the measure performed by N. Based on this, P builds a region, compute a radius and checks if at least one of its children falls within the region. If so, P sends N a `TRY` message containing the children list and the radius. Figure 4.4(b) illustrates this aspect. When N receipts the `TRY` message, it measures its distance to some of the nodes in the list. N tests then if a node falls within the region it builds with the radius. In figure 4.4(c), A falls within the region built by N. N is going to begin a new Join Procedure with A as potential parent. N is said to *go down one layer*. We note that P doesn't keep any information about N.



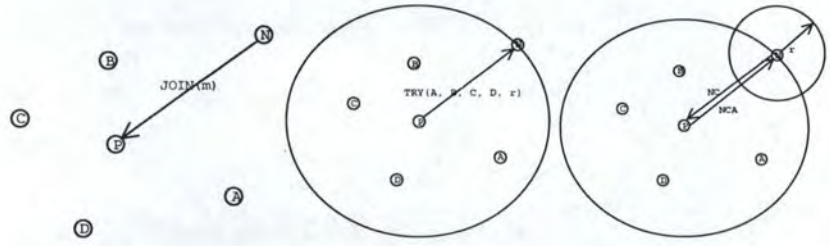
(a) JOIN message (b) TRY message (c) Next iteration

Figure 4.4: Application Level Clustering - Join Procedure (second case)

Figure 4.5 illustrates the second case. The first two steps (figure 4.5(a) and 4.5(b)) are totally identical to the other case. Figure 4.5(c) shows that all the nodes considered are outside N's region. Thus, N is becoming a new child of P. N sends a `NC` (NEW_CLUSTER) message to P. P acknowledges by sending back a `NCA` message.

An example of tree construction will be presented in chapter 6.

For space reasons, figure 4.6 shows only a simplified version of the Finite State Machine (FSM) of ALC. A more complete version is presented in appendix A, section A.4.17. This FSM represents the interaction of a node with its neighborhood and consists of five states: Init, Wait, Connected, Maintenance and Finish. The *Init* state is the initial state. This state marks the beginning of the Join Procedure. Only an `OBJRSP` message can be accepted in this state. The `OBJREQ` message is sent to the potential parent (the first time, it is the root) to know its measurement address. The `OBJRSP` message is the response to the `OBJREQ` message. With the measurement address, the new comer can measure its distance to the potential parent. The *Wait* state is an intermediate state. It is between a not connected state (Init) and a



(a) JOIN message (b) TRY message (c) New cluster

Figure 4.5: Application Level Clustering - Join Procedure (third case)

connected state (Connected). This state manages the Join Procedure as described above. The *Connected* state is the state in which the new comer is connected to the tree. If the node is the root of the tree, it directly goes to this state. In the *Maintenance* state, the node performs a Maintenance Procedure, as described in section 4.5. Finally, the *Finish* state is the final state, i.e. the state after leaving the tree. In this state, the node cannot accept any messages.

A Maintenance Procedure is necessary to make sure that the current hierarchy is still the best one. This procedure is described in section 4.5.

A heartbeat mechanism, described in section 4.6, is also necessary to detect crashes and to recover the lost children.

It results from this mechanism a tree rooted at the source. This multicast tree is uni-source and decentralized. We choose to base the message exchanges on TCP. The transport protocol used for data transporting will be application-specific.

4.3 Tree Building Control Protocol

This section describes the Tree Building Control Protocol (*TBCP*) designed to build overlay trees among participants of a multicast session without any knowledge of the network topology neither any prerequisite knowledge of the full group membership [7].

TBCP is a distributed overlay spanning tree building protocol, whose purpose is to place members in the most optimal position at joining time.

A new node joins the tree at the root. The new comer thus only needs the (SP, P) information, where SP is the IP address of the root and P the port used by the root for the TBCP message exchanges.

Each TBCP node fixes a maximum number of "children" it accepts to accommodate with. This value is called the *fanout* and is used to control the traffic load.

The Join Procedure is a recursive mechanism and works as follow: a new comer (N) contacts a potential parent (P) by sending an `HELLO` message (for the first Join Procedure, N starts to contact the root of the tree). P acknowledges immediately by sending back an `HELLOACK` message containing its children list. For consistency reasons, P can process only one Join Procedure at a time. It thus starts a timer. During this timer, nobody can connect to P. If there is a connection attempt during the timer (i.e. P receives an `HELLO` message), P

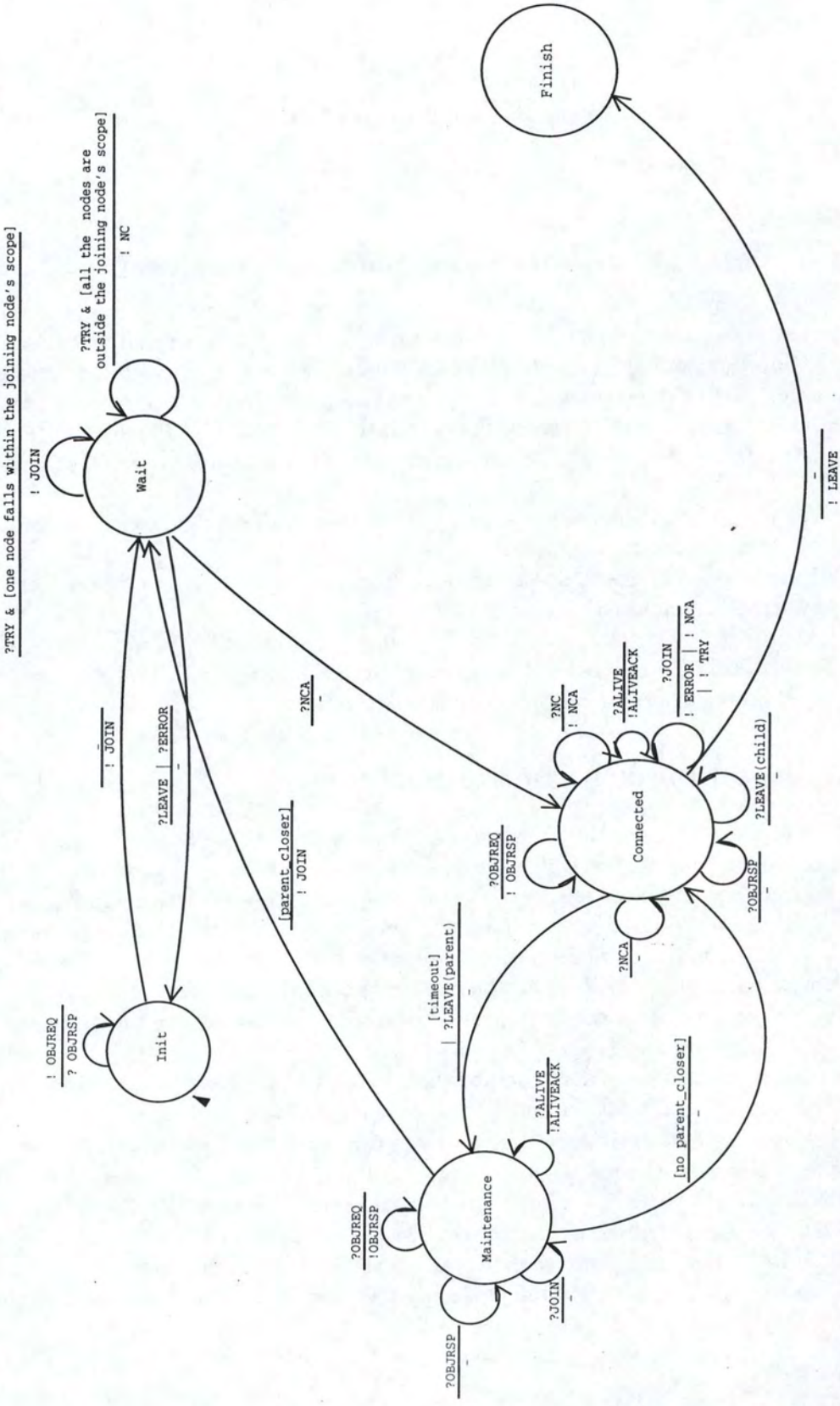


Figure 4.6: Simplified version of ALC Finite State Machine

sends back an ERROR message.

N measures its distance from P and all the nodes in the list (let's call them C_i) and sends this information to P in a JOIN message. If P hasn't received this message within the timer, it sends to N a REJECT message, indicating that N has to restart the Join Procedure since the beginning.

P tries to find a place for N by evaluating all the possible configurations. Figure 4.7 illustrates this mechanism. P tests each configuration, simulating the fact that N or one of the C_i (P's children) has to go down one layer. A score function is used to estimate "how" good each configuration is. The function is based on the distance estimated among P, N and $C_i s$. The score function implemented is the following:

score function = $\max_{\forall M \in \{C_i\} \ U \ N}$ D(P, M), where $\{C_i\}$ is the set of P's children, N the new comer and D(i, j) is the distance between node i and node j along the tree. The chosen configuration is the one with the smallest score.

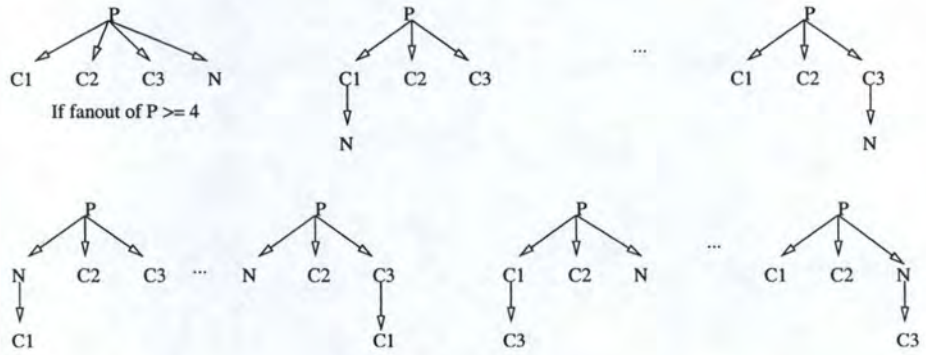


Figure 4.7: Local configuration test

Here, two cases are possible:

1. P accepts N as a child (the fanout is not reached yet).
2. P doesn't accept N as a child (the fanout is reached or the measures towards P are too bad).

In the first case, P sends a WELCOME message to N, which N acknowledges immediately by sending back a WELCOMEACK message. See figure 4.8 for the message exchanges.

In the second case, N or any P's children (say C_j) has to be redirected to another P's child (say C_i). P sends to C_j or N a GO(C_i) message, indicating that the receiver has to go down one layer, which is immediately acknowledged by sending back a GOACK message. A new Join Procedure will now start with C_j or N as potential child and C_i (the new rendez-vous point) playing the role of P. See figure 4.9 for the message exchanges. This figure considers the case where C_j has to go down one layer.

Again, for space reasons, figure 4.10 shows a limited version of the FSM. A more complete version is available in appendix B, section B.4.21. This FSM represents the interactions between a node and its neighborhood and consists of six states: Init, Wait, Connected, JoinProc, Maintenance and Finish. The *Init* state is the initial state. It marks the beginning of the Join Procedure, as described above. None messages can be accepted in this state. The *Wait* is the

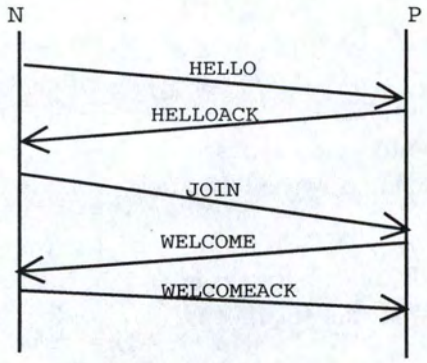


Figure 4.8: Tree Building Control Protocol - Join Procedure (first case)

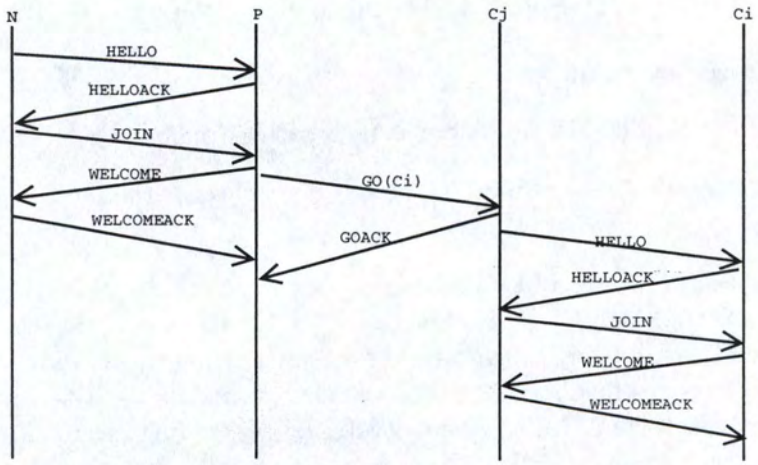


Figure 4.9: Tree Building Control Protocol - Join Procedure (second case)

intermediate state, between a not connected state (Init) and a connected state (Connected). If the node receives a `WELCOME` message, it goes to the connected state. If it receives a `GO` message, it returns to the Init state and has to restart a new Join Procedure. In the *Connected* state, the node is connected to the tree. It the node is the root of the tree, it directly goes to the Connected state. In the *JoinProc* state, the node performs a Join Procedure, acting as a potential parent. In this state, a node can handle only one Join Procedure at a time. In the *Maintenance* state, the node performs a Maintenance Procedure, as described in section 4.5. Finally, the *Finish* state is the final state, i.e. the state after leaving the tree. In this state, the node cannot accept any messages.

As ALC, it results from this mechanism a tree rooted at the source. The multicast tree is uni-source and totally decentralized. We also choose to base the message exchanges on TCP. The transport protocol used for data transporting will be application-specific.

As ALC, TBCP also needs a Maintenance Procedure (see section 4.5) and a heartbeat mechanism (see section 4.6).

4.4 ALC - TBCP comparison

In this section, we try to highlight the big theorical differences between the protocols implemented at Lancaster University. The comparison will focus on three key aspects of the overlays implemented. These key aspects are: the *philosophy* of the tree built, the acceptance level of a new comer in the tree and the way of accepting a new comer.

The philosophy of the tree construction is totally different in both protocols. ALC aims to build a non-constrained tree. So, a node can have an unlimited number of children. This could lead to a tree having only two levels: the root (level 1) and its children (level 2). In opposite, TBCP tries to build a constrained tree. In this tree,a node can have a maximum fixed number of children (cfr the fanout). This number will be application-specific. This could lead to a not very optimal tree. For example, when the fanout is reached, a node has to go down one layer. The parent tests all the possible configurations and chooses the best one but the resulting configuration could be less efficient than the configuration with a higher fanout.

The acceptance level of a new comer in the tree is also different in both protocols. With ALC, it is the new comer that chooses its place in the tree. A node has no power on the choice of its children. This could lead to problems when an overloaded node "accepts" an additional child. In opposite, with TBCP, it is the parent that chooses to accept or not a new comer. A node has a total control on its children.

The way of accepting a new comer is based on two concepts totally different. ALC uses the concept of zone or region, as described in section 4.2. TBCP is based on a score function, as described in section 4.3.

4.5 Maintenance Procedure

The previous sections introduced the Join Procedure for both protocols but it is obvious that nodes join and leave the tree dynamically. This has an impact on the structure of the tree.

After a while, the current position of a node may not be the best one. That's why each node is going to find periodically a better place in order to make the tree more efficient. The

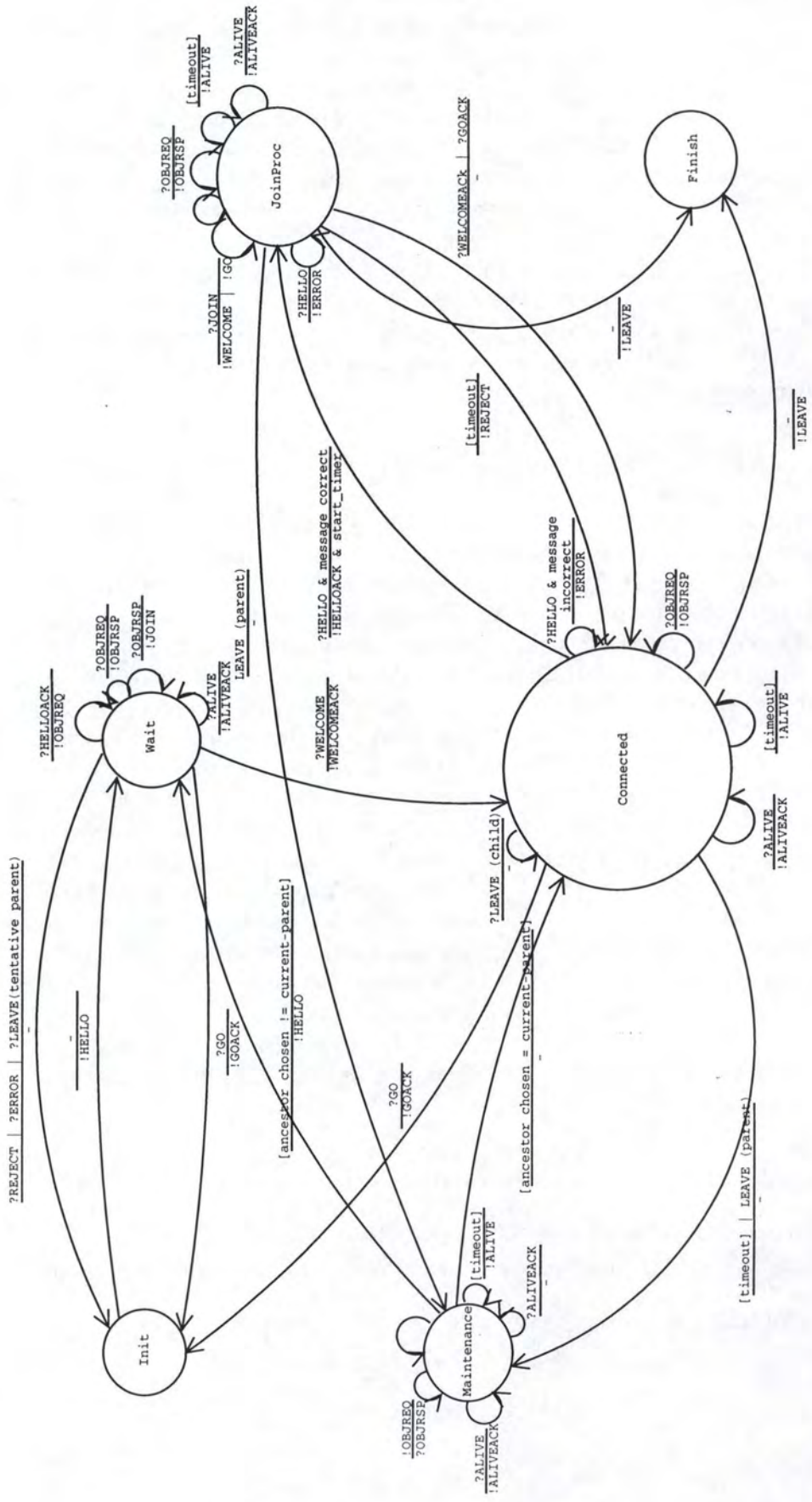


Figure 4.10: Simplified version of TBCP Finite State Machine

idea of the Maintenance Procedure is based on the concept of *root path*. The root path of a node is the path from the root of the tree to that node.

The higher a node is in the tree (i.e. closer to the root), the more root paths it is likely to appear in. So, to avoid the overloading of such nodes, we try to bias the choice of ancestor towards nodes that are lower in the hierarchy. To achieve this, if the root path of a node is R=A_1, A_2, ..., A_n, ancestor A_i will be chosen with probability $\frac{2i}{n(n+1)}$.

If at the end of a maintenance procedure a node has discovered a potential parent closer than its current one, it then moves to that parent, starting a new Join Procedure, as described above. Thus, a node can change place, i.e. be in another branch of the tree. Otherwise, it remains at its place in the tree. A maintenance timer schedules the next maintenance timeout. This maintenance interval is given by $\min[B(1+\delta)^i, M]$, where B is the minimum maintenance interval, M is the maximum maintenance interval, δ is a scaling factor ($\delta > 0$) and i is the number of previous and consecutive maintenance procedures that didn't result in a move of a node. If a node doesn't move, the new maintenance timer will be higher than the previous one. Thus, more a node performs consecutive maintenance procedure that does not result in a move, less maintenance procedure it performs.

The pseudo-code 2 gives the pseudo-code for the ALC implementation of the Maintenance Procedure. The Maintenance Procedure is the same for TBCP. The algorithm works in seven steps.

The first step checks if the node is the root of the hierarchy. If it is, the maintenance procedure has no sense.

The second step aims to determine the position (*positionHierarchy*) of the node in the hierarchy. This is done by computing the number of nodes in the root path.

The third step computes the probability of each ancestor in the root path, starting with the lowest in the hierarchy. Thus, it implements the $\frac{2i}{n(n+1)}$ formula, where n is substituted by *positionHierarchy*.

The fourth step aims to choose the ancestor to eventually join. First, we determine a random number. An ancestor will be chosen if and only if the random number is less than its probability. If not, we subtract the probability of the ancestor to the random number and we test with the next ancestor. As the root probability is the lowest and the current parent probability is the highest, we have to begin the comparison with the current parent. If the random number is not less than the current parent probability, it will never be less than the probability of another ancestor because the current parent probability is the lowest. That's why we subtract the probability of the not chosen ancestor to the random number.

The fifth step finds the PeerState of the chosen ancestor. The *PeerState* is a Java class that contains the state informations a peer retained about another peer (for details, see section 5.2.1.1 and appendix A). This state information contains, among others, the three addresses used by a node: the signalling address (used for the messages exchange), the measurement address (used to perform measure) and the data address (used to transfer data). Thus, the PeerState contains all the informations needed to perform a Join Procedure.

The sixth step checks if the chosen ancestor is the current parent. If so, we just have to restart the maintenance timer. Otherwise, we go to the last step of the algorithm.

The seventh step first checks if this node has a measure against the chosen ancestor. If not, this node first perform a measurement. Next, it compares the measures. There is a timer associated to each measure. When the timer is out, a new measure is performed. This allow a node to keep the latest measure to each node in its neighborhood. If the distance to the chosen ancestor is less than the distance to the current parent, this node has to move. Otherwise, it

Pseudo-Code 2: Pseudo-code for the maintenance procedure

```
//Test if this node is the root.
begin step 1:
   if (root)
      return ;
end step 1

//Determine the position of this node in the hierarchy.
begin step 2:
   state = maintenance;
   int positionHierarchy = 1;

   //Get the parent of this node.
   PeerState peer = parent(this);

   while (peer ≠ null)
      {
         if (peer == root)
            break;

         //Get the parent of the peer.
         peer = PARENT(peer);
         positionHierarchy++;
      }
   end step 2

   //Compute the probability of each ancestor.
   begin step 3:
      double tabProba = new double[positionHierarchy];
      int j = 0;
      int i = positionHierarchy;

      while (j < positionHierarchy)
         {
            tabProba[j] = 2*i / positionHierarchy*(positionHierarchy+1);
            i++;
            j--;
         }
   end step 3

   //Choose the ancestor.
   begin step 4:
      double rand = RANDOM();
      j = 0;

      while (j < positionHierarchy)
         {
            if (rand < tabProba[j])
               break;
            rand = rand - tabProba[j];
```

```
            j++;
        }
end step 4

//Find the PeerState of the chosen ancestor.
begin step 5:
    //Get the parent of this node.
    peer = PARENT(this);
    i = 1;
    int stop = positionHierarchy - j ;

    while (i < stop)
        {
            //Get the parent of peer.
            peer = PARENT(peer);
            i++;
        }
end step 5

//Test if the chosen ancestor is the current parent.
begin step 6:
    if (peer = PARENT(this))
        goto restart;
end step 6

//Compare the measures.
begin step 7:
    if (¬∃ peer.dist)
        PERFORMMEASURE(peer);

    if (peer.dist < PARENT(this).dist)
        {
            start = Wait;

            //Signal its departure to its neighbourhood. LEAVETHEHIERARCHY();

            //Start a Join Prcedure and consider peer as the potential parent.
            STARTJOINHIERARCHY(peer);
        }
    else
        {
            label:restart;
            state = Connected;
            numberMaintenanceWithoutMove++;
            double x = (1 + δ)^numberMaintenanceWithoutMove;
            double T = MIN(B+x, M);
            double latence = T + T/2 + RANDOM(0, T);
```

```
                //Restart the maintenance timer.
                RESTARTMAINTENANCETIMER(latence);
            }
        end step 7
```

has to restart the maintenance timer.

4.6 Heartbeat Timer Negotiation

Because a node can die or be unreachable due to network failures, it is necessary to define a heartbeat mechanism. We describe here the negotiation of the heartbeat timer during the ALC Join Procedure[2].

When a parent P sends to its new child N a `WELCOME` message, P inserts in it a timer information. This information represents the minimum timer value for the `ALIVE` message sent *from N to P*. The maximum value is implicit: if T is the minimum timer value, $T + \frac{3T}{2}$ is the maximum.

N notes this information. In the `WELCOMEACK` message, N adds a timer information too. It has the same meaning as the one in the `WELCOME` message: the minimum timer value for the `ALIVE` message sent *from P to N*.

For each node, the timer value used will be chosen like this: the minimum timer value T plus a value randomly chosen in $[\frac{T}{2}, \frac{3T}{2}]$.

If a node does not reply to an `ALIVE` message, two cases are conceivable:

1. one of the node's children doesn't reply.
2. the current parent doesn't reply.

If a child does not reply, there is nothing special to do. The node has just to consider that its child left the tree.

If the current parent does not reply, the node has to start a join procedure, considering its grand-parent as the tentative parent. This solution is feasible because each node has its root path.

4.7 Limitations

Both protocols defined above are not yet complete. They have five main drawbacks.

First, the protocols are not really fault tolerant, i.e. it does not exist any mechanism to recover from a failure of the root. If the root dies, the tree (or the hierarchy) becomes unstable and, generally, all the nodes die within the minute following the root failure. This drawback will be fixed in further versions of both protocols. A solution could be the root replication. As for Overcast, we could imagine that the three first levels of the hierarchy have degree one and these nodes are a replication of the root. If the root dies, the node in the next level becomes the root. This solution will be easy to implement in TBCP by simply fixing the fanout to one for the three first levels. It will be more difficult for ALC because of the non-constrained aspect of the tree. We could solve this problem in ALC with a new kind

[2]The idea is the same for the TBCP Join Procedure

of object: ROOTREPLICATION. This object will contain an integer indicating the "substitution priority" in case of root failure. The object will be optional and sent by the root/parent to its child in the NCA message. Figure 4.11 shows the resulting tree of this mechanism. From the root to $child_1$, the root replication number will be *1*, meaning that, in case of failure of the root, $child_1$ will become the root. From $child_1$ to $child_2$, the replication number will be *2*, indicating that in case of failure of the root and $child_1$, $child_2$ will become the root. From $child_2$ to $child_3$, the replication number will be *3*, indicating that in case of failure of the root, $child_1$ and $child_2$, $child_3$ will become the root.

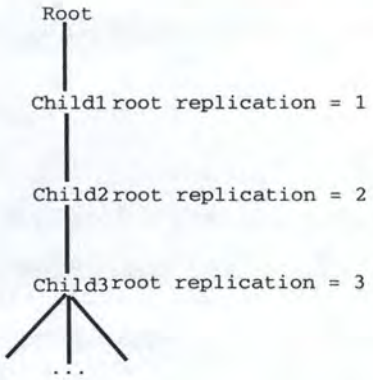


Figure 4.11: Root replication in ALC

Second, there is a problem if a node stays behind a Network Address Translator (NAT). This node has a private address and the NAT translates it dynamically. Thus, the address encoded by the node in the message is the private one. Again, this drawback will be fixed in further versions of both protocols.

Third, there is no error management, i.e. a message received in a wrong state or with a unknown format will simply be ignored.

Fourth, there is the firewall problem. A node behind a firewall will not necessary be seen by other participants. This problem can be solved if the user configures manually its firewall to open all the ports used by the protocols.

Last, there is a problem with the root path update. When a node leaves the tree, it sends a LEAVE message to its neighborhood (parent and children) but the rest of the tree/hierarchy is not aware of this departure. So, when a node performs the maintenance procedure and chooses to join the node that left, it can't because it has disappeared. In the both protocols, none mechanism has been expected to avoid this problem. The problem is similar when a node dies or when a node completely changes its position in the tree after a maintenance procedure. A procedure should be defined to propagate changes in the tree, as the Up/Down protocol in Overcast [5].

The next subsections will examine the problems peculiar to each protocol.

4.7.1 ALC limitations

The main problem of ALC concerns the non-constrained nature of the tree. The tree can have the shape of a "string" or the root can have a large number of children. This can lead to a not so good traffic load management.

In a personal point of view, the fact that a node can not refuse a node as its child is a bad idea. This can lead to an unnecessarily overloading of a node. The overlay protocol should be as light as possible to allow an easy data transmission.

4.7.2 TBCP limitations

A node can process only one join procedure at a time. This can increase needlessly the time needed to find a place in the tree.

The fact that the tree is constrained can lead to a not very optimal tree. A node has to go down one layer if the fanout is reached but the configuration could be more efficient if the fanout was higher.

4.8 Summary

As a summary of this chapter, we add the both protocol described above in the table used at the end of the previous chapter. The structure of the table 4.1 is the same than the previous table. The *TCP* and *UDP* columns indicate if the protocols are based on TCP or UDP. The *centralized* column indicates if the overlay provide a centralized solution or not for tree computing. The *uni-source* column indicates if the tree build by the overlay protocol is uni-source or not. The column *Full knowledge* indicates if, in the tree, there at least one node that has a full knowledge of the topology. The *Interdomain* column indicátes if the overlay is used for interdomain traffic or not. The *Tree* column gives an information about the type of tree build by the overlay protocol. The last column (*Used for*) indicates the utilization of the overlay by an application. The *?* indicates that the information is not given by the authors. For ALC and TBCP, the TCP and UDP columns are put to *Y* because both protocols use TCP for messages exchange and UDP to perform RTT measure.

Overlay	TCP	UDP	Centralized	Uni-source	Full knowledge	Interdomain	Tree	Used for
ALMI	Y	Y	Y	N	Y	N	Shared tree	Multicast sessions where there is a great number of small groups
Narada	?	?	N	N	Y	N	Shortest path	Small and sparse group
Overcast	Y	N	N	Y	?	N	Distribution tree rooted at the source	On-demand and live data delivery
Pastry	?	?	N	N	N	N	Circular name space	P2P application
Yoid	Y	Y	N	Y	Y	N	Mesh topology and shared tree	Replication, data distribution
Scribe	Y	N	N	N	N	N	Multicast tree	Multicast
RON	N	Y	?	?	?	Y	?	Monitor the functioning and quality of the Internet paths
ALC	Y	Y	N	Y	N	N	Cluster hierarchy	video-conference, multiparty games
TBCP	Y	Y	N	Y	N	N	Tree rooted at the source	video-conference, multiparty games

Table 4.1: Application-level overlays overview summary

Part III

Evaluation

Chapter 5

Implementation Details

In this chapter, we describe the architecture (packages, ...) of the ALC and TBCP implementations. First, we detail the structure of both protocols (section 5.1. The code is also introduced by presenting packages, classes, fields and methods (section 5.2). Second, we explain the problems we had to solve while writing the Java implementation (section 5.3). Finally, we propose a way to transmit data above the tree built (section 5.4).

5.1 General Structure

A node in ALC and in TBCP consists of four application-defined components, as shown in figure 5.1.

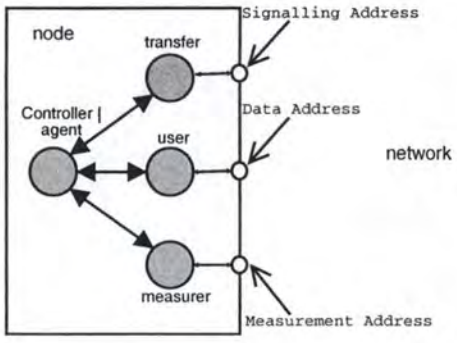


Figure 5.1: Node Architecture

The first component, called *Agent* in ALC and *Controller* in TBCP, is the most important because it manages the protocol. It represents thus the logical behavior of a node, as described in chapter 4. This component handles messages and maintaines states about itself and about nodes with which it is in contact. This component relies on an application-specific signalling address which allows the Agent/Controller of other peers to communicate. The Agent/Controller interacts with other peers via the Transfer component. The Agent/Controller represents the main activity of a node.

The second component is the *User*. The User instructs the Agent/Control
ler to join (as root or as normal node) or leave the tree. In ALC, the User indicates to the Agent how many "best neighbors" it is interested in and in, TBCP, it indicates to the

Controller the fanout. The User provides the Agent/Controller with an application-specific data address which, in combination with the signalling address of the Agent/Controller, allows the Users of other nodes to communicate in an application-specific manner, i.e the way the application using the ALC/TBCP package decides to send data over the tree built by the overlays. In return, the User in ALC will be informed of the best neighbors, including their data addresses. The User component of both protocols will receive similar information about the current parent of the node. The User is the starting point (join) and the ending point (leave) of the actions of the Agent/Controller in the tree. The User is the component that allows the dialogue between the ALC/TBC package and the application using it.

The third component is the *Measurer*. The Measurer provides the Agent/Controller with an application-specific measurement address, which, in combination with the signalling address of the Agent, allows the Measurer of other peers to communicate in an application-specific manner, i.e. send/receive ping messages used to measure the RTT, as defined in chapter 4. The Agent/Controller will instruct the Measurer to obtain a measurement about other node by providing that node's measurement address. The Measurer calls the Agent/Controller back in order to provide the measurement.

Measurements are opaque abstract types from the Agent/Controller's point of view. The Measurer allows the Agent to determine the ordering of two measurements. In ALC, the Measurer allows to determine a region and a radius from a given measurement.

The last component is the *Transfer* component. The Transfer hides the details of communication of messages between peers. It will keep associations between per-node state maintained by the Agent/Controller, and the communication channels maintained by the Transfer component (and its remote counterparts).

Each logical component defined above is represented by a Java package.

5.2 Introduction To The Java Code

The previous section gave an abstract view of the code design. This section presents each component in more details and in a more concrete way, i.e. Java oriented.

5.2.1 Application Level Clustering

ALC is composed of four packages: the Agent package, the Measurer package, the User package and the Transfer package.

5.2.1.1 The `UK.ac.lancs.Clustering.Agent` package

Figure 5.2 details the content of the Agent package. Each box represents an interface or a class. This package is divided in two parts: the data structure (above) and the logical behavior (below). As shown in the figure, some boxes are linked by arrows in the logical behavior. The box above represents an *interface* and the box below its implementation.

The data structure part 12 classes: AgentState, AppData, Message, MessageContent, MessageContentVector, Objet, PeerState, PeerStateVector, SigAddr, TimedAppData, UnknownMessageException and UnknownObjectException.

The *AgentState* class is illustrated in figure 5.3. This class implements the state maintained by the Agent about itself and about its participation in the tree. The AgentState contains the following informations:

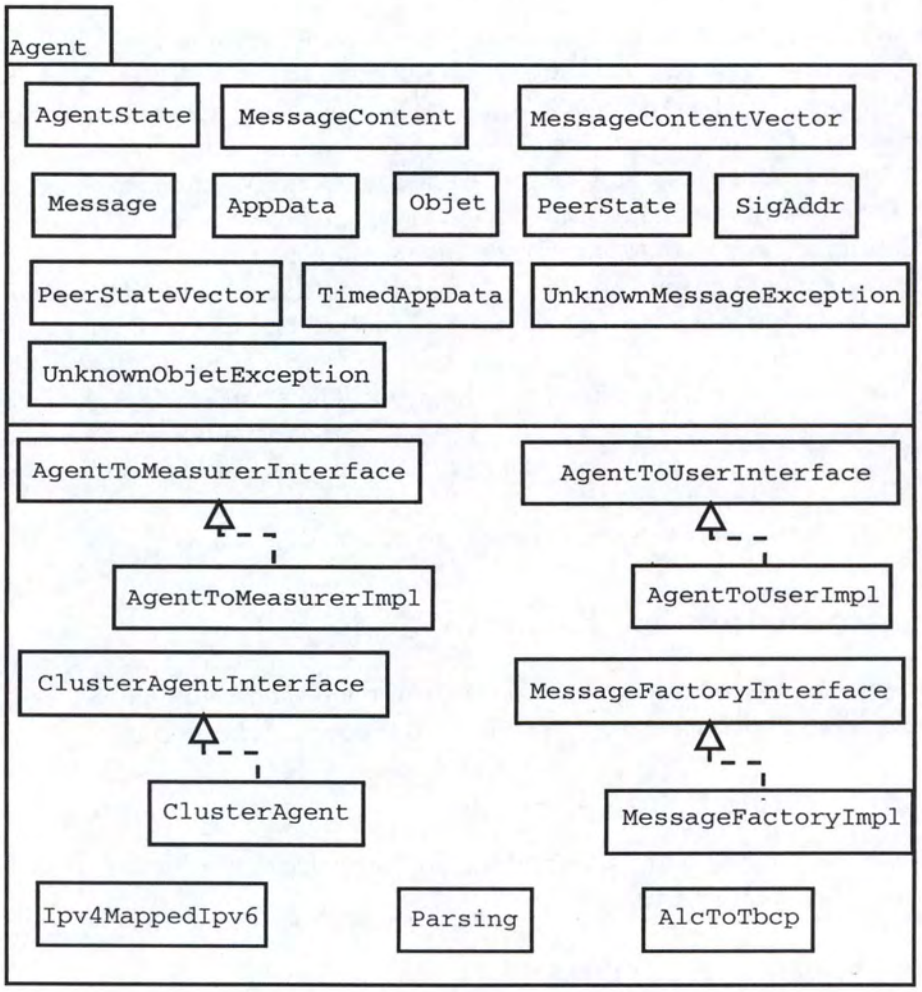


Figure 5.2: The Agent Package

sigAddrAgent: the signalling address of the Agent.

maddrInfoAgent: the measurement address of the Agent.

daddrInfoAgent: the data address of the Agent.

rootAddress: the signalling address of the root of the tree.

peers: the state of each peers with which the Agent is in contact.

siblingsPending: the peers that are potential parents and have no recent measurement.

siblingsMeasured: the peers that are potential parents and have recent measurement.

currentParent: the state of the current parent.

tentativeParent: the peer that the Agent considers as a prospective parent.

children: the peers registered as children of the Agent.

All these fields are declared *private*. They can be accessed through accessors (*set* and *get* methods) but they are not shown in figure 5.3 for readability and space reasons. In the remainder of this chapter, we will not present the accessors and the constructors for the same reasons.

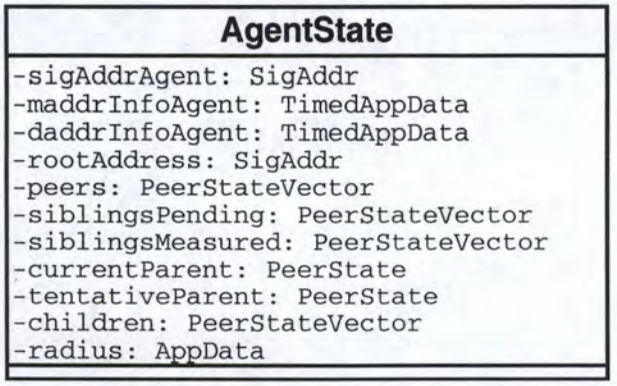


Figure 5.3: The AgentState class

The *AppData* class (figure 5.4) represents a sequence of octets with an application-specific meaning. It is used to encode as byte streams more complex data structures like measurements, radius and addresses. The sequence of octets is represented as a byte array (*info*) and is declared as *private*. The *isLessThan* method tests if *this* appdata is less than *app*.

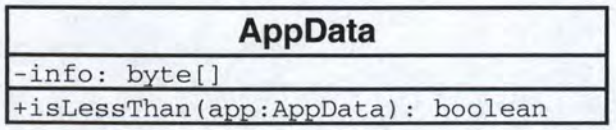


Figure 5.4: The AppData class

The *Message* class (figure 5.5) gives an abstract view of the messages, i.e. only the message identifier. The fields are declared *public*.

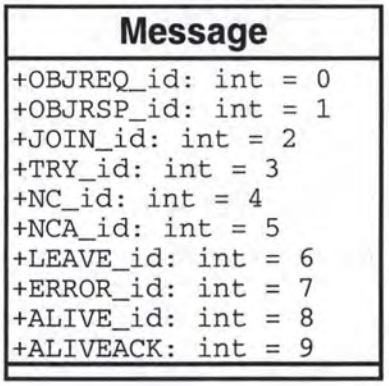


Figure 5.5: The Message class

The *MessageContent* class (figure 5.6) represents the content of messages exchanged between nodes, i.e. all the objects in their Java representation that can be found in a message. All the fields are declared *private*. This class is used in combination with the *MessageContentVector* class that gives a list of MessageContent object. These two classes will be used during the parsing of messages to convert the messages in a byte array format into a MessageContentVector object that is easier to handle.

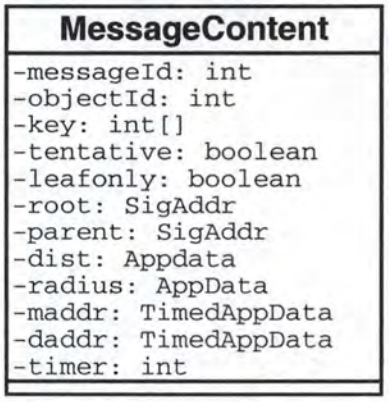


Figure 5.6: The MessageContent class

The *Objet* class (figure 5.7) gives a view of the objects, i.e. their identifiers and their Java representation. Objects are components of messages. All the fields are declared *public*. This class is essentially used by the *MessageFactoryImpl* class to include the object in the messages.

The *PeerState* class (figure 5.8) represents the state information that the Agent maintains about the peers with which it is in contact. The Agent keeps the following informations:

SigAddrPeer: the signalling address of the peer.

MaddrInfoPeer: the measurement address of the peer.

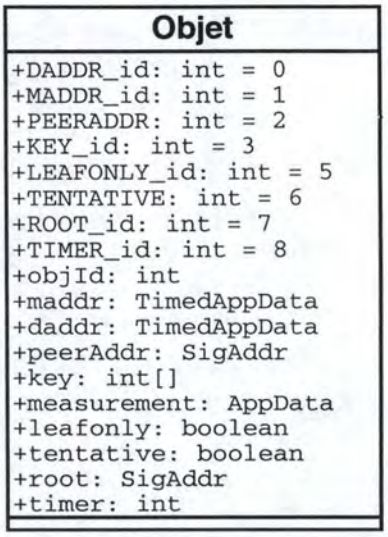


Figure 5.7: The Objet class

DaddrInfoPeer: the data address of the peer.

Mode: the "state" in which the Agent is in relation to this peer. The different modes are: *Error* (the Agent sent a `ERROR` message to this peer), *boring* (the default mode), *Measuring* (the Agent is performing a measurement to this peer), *Tentative* (the Agent acts as a tentative peer to this peer) and *Discovering* (the Agent performs a Join Procedure with this peer as potential parent).

LocalDist: the latest measurement made by the Agent against this peer.

RemoteDist: the latest measurement made by this peer against the Agent.

Leafonly: indicates whether the Agent will accept children or not.

ParentPeer: the peer which is the parent of the Agent.

Timeout: a period of time that the Agent will wait for a response before retransmitting the message.

Lives: the number of times a message will be retransmitted without a response.

HeartbeatTimer: the minimum period of time that the Agent will wait before sending an `ALIVE` message to this peer.

AliveAckWanter: indicates that an `ALIVE` message has been sent to this peer and that an `ALIVEACK` message is expected.

JoinRspPending: indicates that a `JOIN` message has been sent to this peer and that a `TRY` or `NCA` message is expected

NcaPending: indicates that a `NC` message has been sent to this peer and a `NCA` message is expected.

DaddrWanter: indicates that an `OBJREQ` message requesting the data address has been sent to this peer and that an `OBJRSP` message is expected.

MaddrWanter:indicates that an `OBJREQ` message requesting the measurement address has been sent to this peer and that an `OBJRSP` message is expected.

All the fields are declared *private*, except the modes that are declared *public*. The four methods presented in figure 5.8 are used to handle the timeout problems. All the actions performed by these methods are described in appendix A. This class should be used in combination with the *PeerStateVector* that gives a list of PeerState objects.

Figure 5.8: The PeerState class

The *SigAddr* class (figure 5.9) represents the signalling address of a node. As a node can be IPv4 stack only, IPv6 stack only or dual stack, this class to store the IPv4 and the IPv6 addresses. All the addresses are stored in an IPv6 format. The fields are declared *private*. The *dualStack* method indicates wether the node is dual stack or not. The *Ipv4Stack* method indicates whether the node contains at least an IPv4 stack. The *equal* method tests if this signalling address equals *sa*.

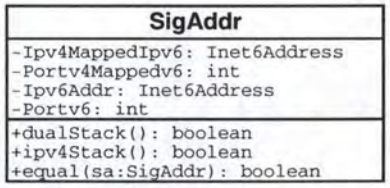


Figure 5.9: The SigAddr class

The *TimedAppData* class (figure 5.10) represents a data structure that is used with the informations stored with a timeout information. The TimedAppData is thus composed of an AppData information plus an expiration time and a boolean indicating if the data can be passed the another peer.

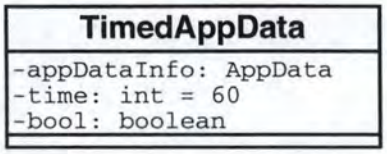


Figure 5.10: The TimedAppData class

The *UnknownMessageException*is thrown by the *parseMessage*() function in the Parsing class when the Agent received a message of an unknown type. The *UnknownObjectException* is thrown by the constructor of the Objet class when it meets an object of an unknown type.

The logical behavior part contains four interfaces and seven classes: AgentToMeasurerInterface, AgentToMeasurerImpl, AgentToUserInterface, AgentToUserImpl, ClusterAgentInterface, ClusterAgent, Ipv4MappedIpv6, MessageFactoryInterface, MessageFactoryImpl, Parsing, AlcToTbcp.

The *AgentToMeasurerInterface* interface (figure 5.11) is used by the Agent to communicate with the Measurer. There are five events from the Agent to the Measurer:

getAddress()⇒(*maddr*)
The Agent needs to know immediately the measurement address of the node.

maddr is a TimedAppData identifying the measurement address of the node.

measure(*peer*, *maddr*)
The Agent needs a measurement from a peer. The response should be through a *measured* event.

peer is a signalling address identifying the peer.

maddr is a TimedAppData identifying the peer's measurement address.

encode(*dist*)⇒(*data*)
The Agent needs a measurement converted to octets for transmission.

dist is the Measurement to be converted.

data is the measurement converted to AppData.

decode(*data*)⇒(*dist*)
The Agent needs a measurement converted from octets after transmission.

data is the AppData to be converted.

dist is the measurement after conversion.

getRegion(*dist*)⇒(*Region*)
The Agent needs to compute the region in which its children must fall to be the potential parent of a prospective peer.

dist the measurement offered by the prospective peer.

Region A region including the best (inclusive) measurement, the worst (exclusive) measurement for suitable children and the worst measurement that the prospective peer should accept from the selected children.

The *AgentToMeasurerImpl* class implements this interface.

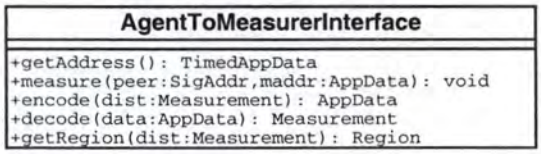


Figure 5.11: The AgentToMeasurerInterface interface

The *AgentToUserInterface* interface (figure 5.12 is used by the Agent to communicate with the User. There are four event types from the Agent to the User:

getAddress()⇒(*daddr*)
The Agent needs to know immediately the data address of the node.

daddr is a TimedAppData representing the data address of the node.

setParent(*peer*, *daddr*)
The Agent has determined its (new) position in the hierarchy, and is identifying the parent. This event occurs whenever the parent changes or when the data address of the parent changes.

peer is a signalling address identifying the parent node.

daddr is a TimedAppData identifying the data address of the parent node.

interestingPeer(*peer*, *daddr*)
The Agent has identified a peer that is among the best n peers requested by the *interest* event, or its data address has changed.

peer is a signalling address identifying the node.

daddr is a TimedAppData identifying the data address of the node.

uninterestingPeer(*peer*)
The Agent has identified a peer that is no longer among the best n peers requested by the *interest* event.

peer is a signalling address identifying the node.

The *AgentToUserImpl* class implements this interface.

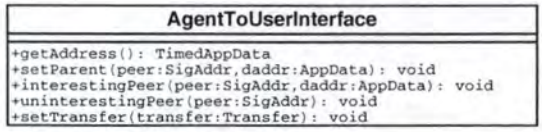


Figure 5.12: The AgentToUserInterface interface

The *ClusterAgentInterface* interface (figure 5.13) represents the Agent. This interface allows the Agent to start/stop its actions in the cluster hierarchy and the processing of the

messages. The *ClusterAgent* class implements this interface. The processing of the messages is performed as described in appendix A.

messageProcessing(*message*, *socket*, *portPeerListen*, *inet*, *port*)
This method allows the processing of messages received from a peer.

message is the message received from a peer.

socket is the socket of the remote host.

portPeerListen is the port used by the remote host to listen/accept incoming connections.

inet is the IPv4 address of the remote host if it is dual stack and if it communicates with its IPv6 channel. Otherwise, it is set to null.

port is the port corresponding to the IPv4 address.

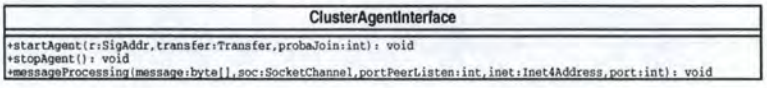


Figure 5.13: The ClusterAgentInterface interface

The *Ipv4MappedIpv6* class allows the transformation of an IPv4 address into and IPv4 Mapped IPv6 address and inversely.

The *MessageFactoryInterface* allows the building of messages of any types, as described in appendix A

The *Parsing* class transforms a message received in byte format into a MessageContentVector object that is easier to process.

The *AlcToTbcp* class allows to transform locally the cluster hierarchy in a constrained tree.

5.2.1.2 The `UK.ac.lancs.Clustering.Measurer` package

Figure 5.14 gives a view of the content of the Measurer package. This package is divided in two parts: the data structure (above) and the logical behavior (below).

The data structure contains three classes: Measurement, Radius and Region.

The *Measurement* class represents a measurement, i.e. the internal representation of a measurement (i.e. RTT) performed by the Measurer with another node. Two methods are defined to transform a Measurement object in an AppData object and inversely.

The *Radius* class (figure 5.16) represents the radius information computed by the Measurer.

The *Region* class (figure 5.17) represents a region, as defined in chapter 4).

The logical behavior contains an interface and four classes: MeasurerToAgentInterface, MeasurerToAgentImpl, Measurer, PerformMeasure, PermanentServerRTT.

The *MeasurerToAgentInterface* interface (figure 5.18) is used by the Measurer to communicate with the Agent. There is only one event that could originate from the Measurer:

measured(*peer*, *dist*, *timer*)
The Measurer is indicating that it has completed a requested measurement with a given peer.

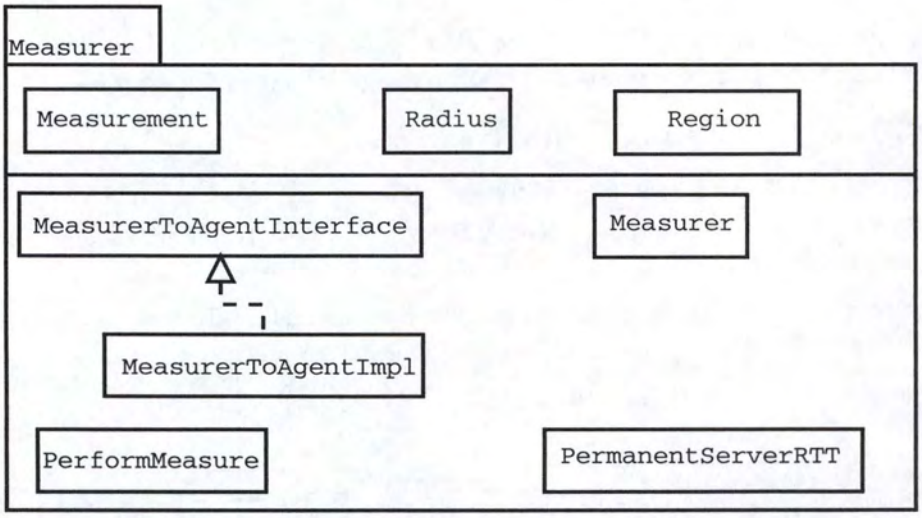


Figure 5.14: The Measurer Package

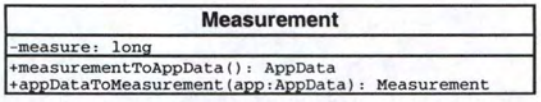


Figure 5.15: The Measurement class

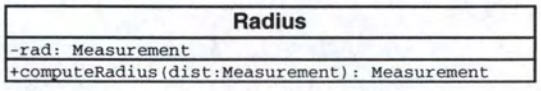


Figure 5.16: The Radius class

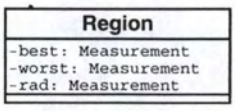


Figure 5.17: The Region class

peer is a signalling address identifying that peer.

dist is the Measurement which the Agent can ask the Measurer to convert into AppData.

timer is an integer representing an absolute value for the measurement timeout, i.e. the time the measurement will during which be valid.

The *MeasurerToAgentImpl* class implements this interface.

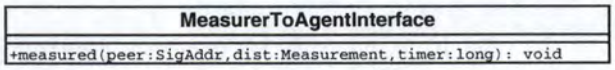


Figure 5.18: The MeasurerToAgentInterface interface

The *Measurer* class (figure 5.19) represents the Measurer component. It determines the measurement address of the node and can be to convert the AppData form of the address into a DatagramSocket.

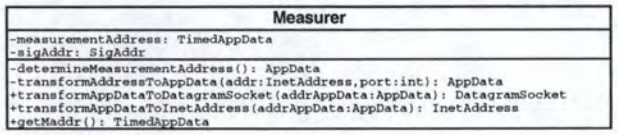


Figure 5.19: The Measurer class

The *PerformMeasure* class allows the Measurer to perform measurement to specified peers. The measurement is a Round-Trip Time (RTT).

Finally, the *PermanentServerRTT* class is used by the Measurer to answer the measurement queries of other peers.

5.2.1.3 The `UK.ac.lancs.Clustering.User` package

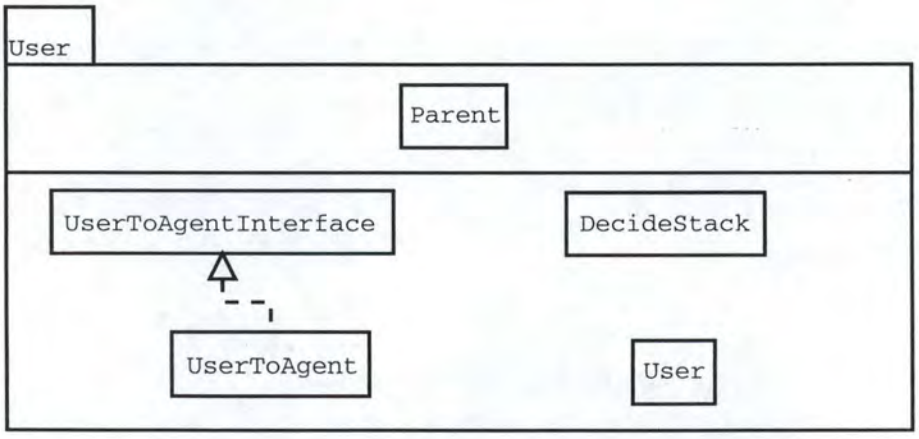


Figure 5.20: The User Package

As shown in figure 5.20, the User package is divided in two parts: the data structure (above) and the logical behavior (below).

The data structure part contains only one class: Parent. The *Parent* class allows the User to keep the informations given by the Agent about the current parent of the node.

The logical part contains an interface and three classes: UserToAgentInterface, UserToAgentImpl, DecideStack and User.

The *UserToAgentInterface* interface (figure 5.21) is used by the User to communicate with the Agent. There are four types of event from the User to the Agent:

join(*peer*, *probaJoin*)
The User wishes to participate in a cluster hierarchy.

peer is a signalling address indicating the root peer.

probaJoin is an integer representing the number used for the probabilistic join.

accept()
The User wishes to be the root of a cluster hierarchy.

leave()
The User wishes to stop participating in a cluster hierarchy.

interest(*count*)
The User wishes to be informed of the data addresses of the best peers in the cluster hierarchy it is participating.

count is an integer identifying the number of ideal peers.

The *UserToAgentImpl* class implements this interface.

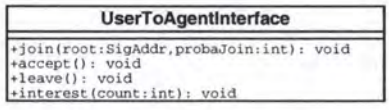


Figure 5.21: The UserToAgentInterface interface

The *DecideStack* class allows the User to decide if the host's network stack is IPv4 only, IPv6 only or dual stack.

The *User* class (figure 5.22) represents the User component. This class also provides the data address of the node. It also keeps information about the current parent and the interesting peers.

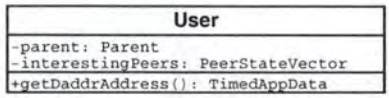


Figure 5.22: The User class

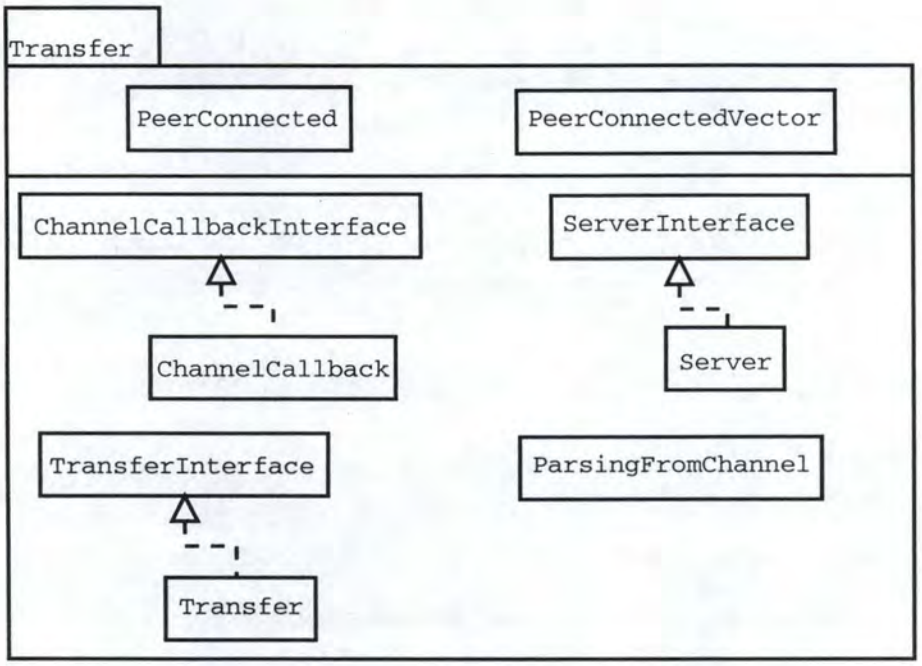


Figure 5.23: The Transfer Package

5.2.1.4 The `UK.ac.lancs.Clustering.Transfer` package

As shown in figure 5.23, the Transfer package is divided into two parts: the data structure (above) and the logical behavior (below).

The data structure part contains two classes: PeerConnected and PeerConnectedVector.

The *PeerConnected* class (figure **??**) is a data structure used by the Transfer component to keep information about the peer which it is in contact with. All the fields are declared *private*. The PeerConnectedVector class gives a list of PeerConnected objects.

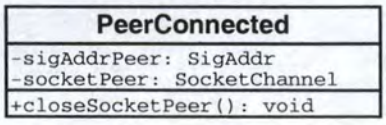


Figure 5.24: The PeerConnected class

The logical behavior contains several interfaces and classes: ChannelCallbackInterface, ChannelCallback, ServerInterface, Server, TransferInterface, Transfer and ParsingFromChannel.

As messages may be arriving from multiple peers/nodes in any order, the *ChannelCallbackInterface* interface defines methods used to be sure that the message read is complete. The *ChannelCallback* class implements this interface

The *ParsingFromChannel* class allows the "cutting" of a message received by the Transfer component in different parts that will be understood by the Agent. This class should be used in combination with the ChannelCallback class.

The *ServerInterface* (figure 5.25) interface defines the behavior of a server used to accept incoming connections, to read messages on socket channels and to send messages on socket channels. This server is based on the *Selector* class that allows to handle multiple I/O operations inside only one thread. The *Server* class implements this interface.

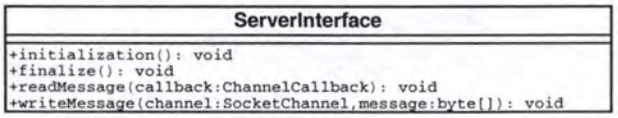


Figure 5.25: The Server interface

The *TransferInterface* interface represents the Transfer component. It is used by the Agent to send data to other peers. There are two methods to send data:

sendMessage(*message*, *receiver*)
This method sends a message to a well-known peer, i.e. a peer already present in the peer list of the Agent.

message is the message to send.

receiver is the signalling address identifying the receiver.

sendMessage(*message*, *receiver*, *port*)
This method sends a message to an unknown peer, i.e. a peer not present in the peer list of the Agent.

message is the message to send.

receiver is the signalling address identifying the receiver.

port is the port used by the receiver.

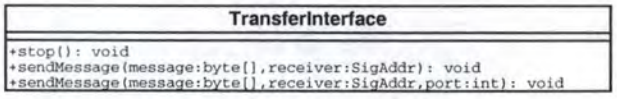


Figure 5.26: The Transfer interface

5.2.2 Tree Building Control Protocol

TBCP is composed of four packages: the Controller, the Measurer, the User and the Transfer packages.

5.2.2.1 The `UK.ac.lancs.tbcp.Controller` package

Figure 5.27 shows the content of the Controller package. Each box represents an interface or a class. This package is divided in two parts: the data structure (above) and the logical behavior (below). As shown in the figure, some boxes are linked by an arrow in the logical behavior part. The box above represents an *interface* and the box below its implementation.

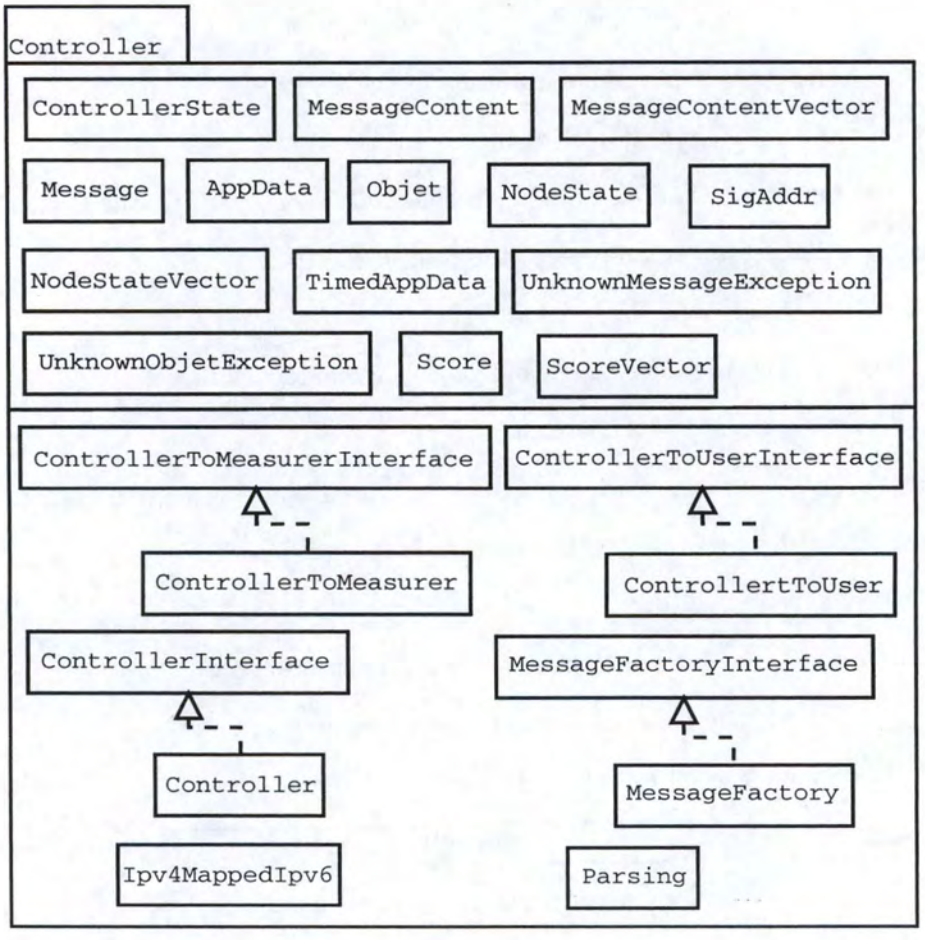


Figure 5.27: The ControllerPackage

The data structure part contains 14 classes: AppData, ControllerState, Message, MessageContent, MessageContentVector, NodeState, NodeStateVector, Objet, Score, ScoreVector, SigAddr, TimedAppData, UnknownMessageException, UnknownObjectException. The AppData, SigAddr, TimedAppData, UnknownMessageException and UnknownObjectException are identical to those used in TBCP.

The *ControllerState* class (figure 5.28) contains the state informations that the Controller maintains about itself and its position in the tree. The Controller keeps the following informations:

sigAddrController: the signalling address of the Controller.

maddrController: the measurement address of the Controller.

daddrController: the data address of the Controller.

fanout: the maximum number of children that will be accepted by the Controller.

rootAddress: the signalling address of the root of the tree.

timerJoin: the timer value for the Join Procedure.

nodes: the list of all the nodes the Controller is in contact with.

currentParent: the state of the current parent.

tentativeParent: the node which the Controller views as a prospective parent.

children: the nodes registered as children of the Controller.

tentativeChild: the node that is a Controller's potential child.

measurePending: the nodes that have no recent measurement.

measurePerformed: the nodes that have recent measurement.

All these fields are declared *private*.

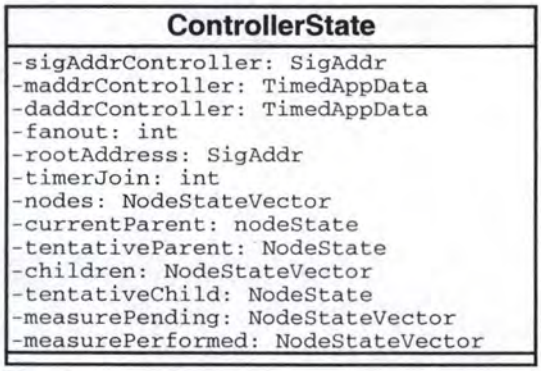


Figure 5.28: The ControllerState class

The *Message* class (figure 5.29) gives an abstract view of the messages, i.e. only the message identifier. The fields are declared *public*.

Figure 5.29: The Message class

The *MessageContent* class (figure 5.30) represents the content of messages exchanged between nodes, i.e. all the objects in their Java representation that can be found in a message. All the fields are declared *private*. This class is used in combination with the *MessageContentVector* class that gives a list of MessageContent objects. These two classes are used during the parsing of messages to transform the message in a byte array format into a MessageContentVector object that is easier to process.

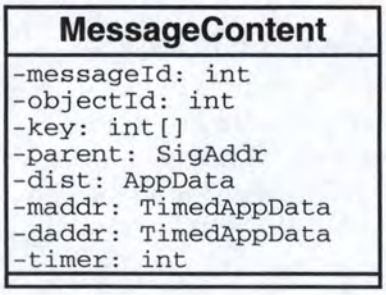


Figure 5.30: The MessageContent class

The *Objet* class (figure 5.31) gives a view of the objects, i.e. their identifiers and their Java representation. Objects are component of messages. the identifiers are declared *public* and the others fields are declared *private*. This class is essentially used by the *MessageFactoryImpl* class to include the object in the messages.

The *Score* class (figure 5.32) represents the informations about a configuration kept by the Controller during the calculation of the score function. The informations are the following:

children is the children list in case of this configuration will be chosen by the Controller.

goDown is the node that will go down one layer if this configuration is chosen.

parent is the parent of the node that will go down one layer.

dist is the distance of this configuration.

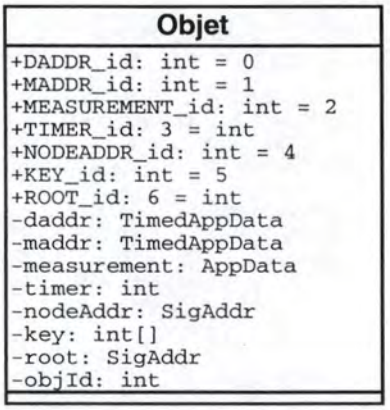


Figure 5.31: The Objet class

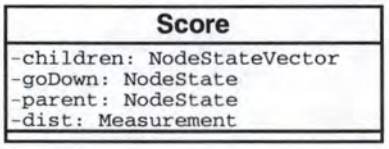


Figure 5.32: The Score class

All these fields are declared *private*.

The logical behavior part contains four interfaces and six classes: ControllerToMeasurerInterface, ControllerToUserInterface, ControllerToMeasyrer, ControllerToUser, ControllerInterface, Controller, Ipv4MappedIPv6, MessageFactoryInterface, MessageFactory and Parsing. The Ipv4MappedIpv6 class is totally identical to the synonym class in ALC. The MessageFactoryInterface interface and the Parsing class are basically the same as in ALC but adapted to TBCP.

The *ControllerToMeasurerInterface* interface (figure 5.33) is used by the Controller to communicate with the Measurer. There are five types of events from the Controller to the Measurer:

getAddress()⇒(*maddr*)
The Controller needs to know immediately the measurement address of the node.

maddr is a TimedAppData representing the measurement address of the node.

measure(*node*, *maddr*)
The Controller asks the Measurer to perform a measurement against a specific node. The measurement will be returned as a *measured* event.

node is a signalling address identifying the node.

maddr is a TimedAppData representing the measurement address of the node.

encode(*dist*)⇒(*data*)
The Controller needs a measurement converted to octets.

dist is a Measurement to be converted.

data is the measurement after the conversion.

decode(*data*)⇒(*dist*)
The Controller needs a measurement converted from octets.

data is the AppData to be converted.

dist is the measurement after the conversion.

compare(*dist1*, *dist2*)⇒(*value*)
The Controller wants to compare to measurements.

dist1 is a Measurement to be compared with *dist2*.

dist2 is a Measurement to be compared with *dist1*.

value is an integer representing the result of the comparison

The *ControllerToUser* class implements this interface.

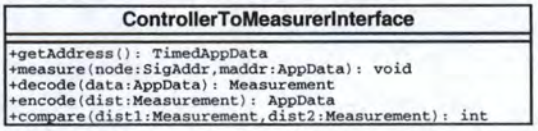


Figure 5.33: The ControllerToMeasurerInterface interface

The *ControllerInterface* interface (figure 5.34) represents the Controller. It allows the Controller to start/stop its actions in the tree and the message processing. The *Controller* class implements this interface. The message processing is performed as described in appendix B.

messageProcessing(*msg*, *channel*, *portNodeListen*, *inet*, *port*)
This method allows the processing of a message received from a node.

msg is the message received from a node.

channel is the channel used to communicate with the remote node.

portNodeListen is the port used by the remote node to accept/listen incoming connections.

inet is the IPv4 address of the remote node if it is dual stack and if it communicates with its IPv6 channel. Otherwise, it is set to null.

port: is the port corresponding to the IPv4 address.

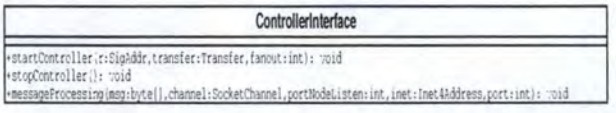


Figure 5.34: The ControllerInterface interface

The *ControllerToUserInterface* interface (figure 5.35) is used by the Controller to communicate with the User. There are two types of events from the Controller to the User:

getAddress()⇒(*daddr*)

The Controller needs to know immediatly the data address of the node.

daddr is a TimedAppData representing the data address of the node.

setParent(*parent*, *daddr*)

The Controller has determined its position (possibly a new one) in the tree. This event occurs either when the parent changes or when its data address changes.

parent is a signalling address indicating the signalling address of the node.

daddr is a TimedAppData representing the data address of the node.

The *ControllerToUser* class implements this interface.

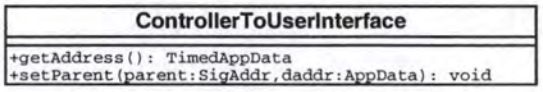


Figure 5.35: The ControllerToUserInterface interface

5.2.2.2 The `UK.ac.lancs.tbcp.Measurer` package

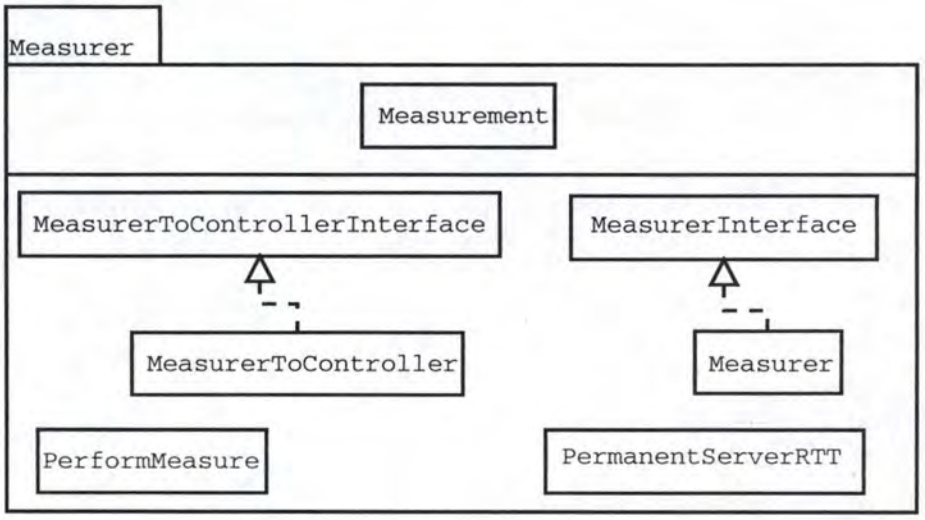


Figure 5.36: The Measurer Package

Figure 5.36 shows the content of the Measurer package. This package is divided in two parts: the data structure (above) and the logical behavior (below).

The data structure part contains only one class (*Measurement*) that is totally identical to the Measurement class in ALC.

The logical behavior contains two interfaces and four classes: MeasurerToControllerInterface, MeasurerToController, MeasurerInterface, Measurer, PerformMeasure and PermanentServerRtt. The last two classes are identical to those in ALC.

The *MeasurerToControllerInterface* interface (figure 5.37) is used by the Measurer to communicate with the Controller. There is only one event from the Measurer to the Controller:

measured(*node*, *dist*)
The Measurer is indicating that it has completed a requested measurement to a given node

node is a signalling address identifying that node.

dist is a Measurement representing the measurement performed that the Controller could ask then the Measurer to convert to AppData).

The *MeasurerToController* class implements this interface.

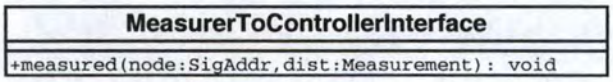


Figure 5.37: The MeasurerToControllerInterface interface

The *MeasurerInterface* interface (figure 5.38) represents the Measurer and determines the measurement address of the node. This interface is implemented by the *Measurer* class.

Figure 5.38: The MeasurerInterface interface

5.2.2.3 The `UK.ac.lancs.tbcp.User` package

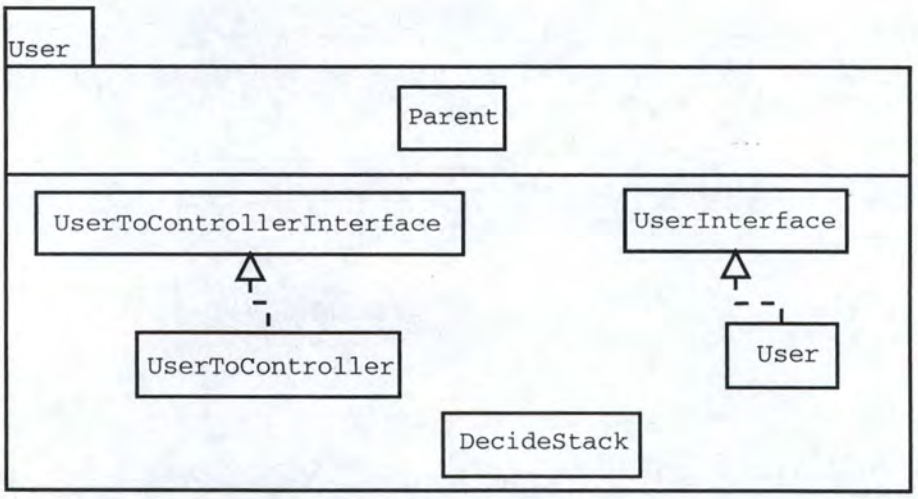


Figure 5.39: The User Package

As shown in figure 5.39, the User package is divided in two parts: the data structure (above) and the logical behavior (below).

The data structure part contains only one class (*Parent*) that is identical to the same class in ALC.

The logical behavior part contains two interfaces and three classes: UserToControllerInterface, UserToController, UserInterface, User and DecideStack. The last class is the same as the one in ALC.

The *UserToControllerInterface* interface (figure 5.40) is used by the User to communicate with the Controller. There three types of events from the User to the Controller:

join(*root*, *fanout*)
The User asks the Controller to join an existent tree.

root is a signalling address indicating the root address.

fanout is an integer indicating the maximum number of children this node might have.

accept(*fanout*)
The User orders the Controller to be the root of the tree.

fanout is an integer indicating the maximum number of children this node might have.

leave()
The User orders the Controller to leave the tree.

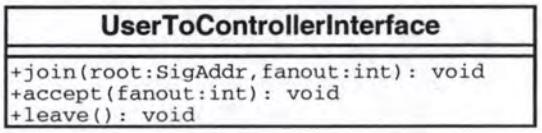


Figure 5.40: The UserToControllerInterface interface

The *UserToController* class implements this interface.

The *UserInterface* interface (figure 5.41) represents the User and determines the data address of the node. This interface is implemented by the *User* class.

Figure 5.41: The User interface

5.2.2.4 The `UK.ac.lancs.tbcp.Transfer` package

The Transfer package in TBCP is totally identical to the Transfer package in ALC (see section 5.2.1.4).

5.3 Implementation Issues

This section presents the problems we had to solve during the Java implementation.

The first problem was the IP stack discovery. We wished that a node could discover by itself its type of IP stack. Therefore, a node could know whether it is dual stack, IPv4 only stack or IPv6 only stack. Our first idea was to open an IPv4 socket and an IPv6 socket on a host and then see the exceptions launched at runtime. This solution was impossible because Java makes transparent the socket opening, i.e. the programmer does not know whether he uses an IPv4 or an IPv6 socket.

Finally, we found the solution in jdk 1.4.1. We used the `NetworkInterface.getNetworkInterfaces()` method from the `java.net` package. This method returns an Enumeration containing all the network interfaces of the host: IPv4 address (if it exists), IPv6 address (if it exists) and the loopback address (v4 and/or v6). Lastly, some handlings (Enumeration, casting, exceptions) were needed to allow to decide the kind of IP stack.

The second problem was the Transfer component, and more specially the Server class. This problem was due to the utilization of the `java.nio` and `java.nio.channel` packages available since jdk 1.4.1.

In the previous versions of the jdk, to run an application that uses several sockets, the programmer had to start a thread for each connection. In spite of that, he could encounter problems like operating system limits, deadlocks, ... Now, the programmers have the opportunity to use the *selector*, a tool allowing to manage several simultaneous sockets in a single thread. The selector performs multiplexing, i.e. managing several I/O actions in a single thread.

The difficulties were the understanding and the utilization of the selector. The solutions were found in [34, 35, 36, 37].

5.4 Data Transmission

This section introduces the way we could transfer data over the tree built by ALC and TBCP. Remember that the purpose of ALC and TBCP is to build a tree to emulate multicast communications. For the moment, nothing is foreseen relating to the data transmission.

We think that the data transfer should be independent of ALC and TBCP. The simpler and more elegant solution is that the application above ALC and TBCP (i.e. the application using them) has to manage the data transfer. This application will need to know the data address of the parent (for the potential acknowledgements) and the data address of the children.

The interactions between the application and the underlying overlay protocol will be possible by using the User. To play this intermediate role, the User needs to know more informations about the Agent/Controller's neighborhood: the parent data address (the User already knows it) and the children data addresses.

Thus, new events between the User and the Agent/Controller (and inversely) should be added. From the User to the Agent/Controller, only one event is enough:

getChildren()⇒*children*

The User needs to know immediately the list of children of the Agent/Controller.

children is a Peer/NodeStateVector representing the list of children.

From the Agent/Controller to the User, two events are needed:

setChild(*sigAddr*, *daddr*)
A new child joined the Agent/Controller. This event happens if there is a new child or the data address of a child has changed.

sigAddr is a signalling address identifying the child.

daddr is the data address of the child.

removeChild(*sigAddr*)
A child left the Agent/Controller.

sigAddr is a signalling address identifying the child.

Chapter 6

Performance Measures

This chapter discusses the performances of the overlays exposed in chapter 4. These performances have been observed on the PlanetLab overlay network.

First, we introduce Planet Lab (section 6.1). Then, we present and discuss the measurements done for Application-Level Clustering (ALC - section 6.2). Next, we introduce the measurements done for Tree Building Control Protocol (TBCP - section 6.3). Finally, section 6.4 concludes this chapter.

6.1 Planet Lab

Planet Lab [38, 39, 40, 41] is a global overlay network for developing and accessing new network services. Their goal is to grow to 1000 geographically distributed nodes, connected by a diverse collection of links. Toward this end, they are putting Planet Lab nodes into edge sites, co-location and routing centers, and homes (i.e. at the end of DSL lines and cable modems). Planet Lab is designed to support both short-term experiments and long-running services. Currently running services include peer-to-peer networks, and content distribution networks.

There are currently more than 115 machines at 45 sites world-wide (see figure 6.1) available. Planet Lab creates an environment in which to conduct experiments at Internet scale. The most obvious is that network services deployed on Planet Lab experience all of the behaviors of the real Internet where nothing (latency, bandwidth, ...) is predictable.

Over the next two years, their intention is to grow Planet Lab from the current 115 nodes into a world-wide federation of 1000 nodes with sites in Universities, co-location centers, industrial labs and even the Abilene (Internet2) backbone routing centers. *Abilene* [42] is an advanced backbone network that connects regional network aggregation points, to support the work of Internet2 universities as they develop advanced Internet applications. *Internet2* [43] is a consortium being led by 202 universities working in partnership with industry and government to develop and deploy advanced network applications and technologies, accelerating the creation of tomorrow's Internet.

Each Planet Lab node consists of a Linux-based PC running specially developed virtual machine technology allowing experiments to be conducted independently. This virtual machine aims to provide better security and resource isolation between services running over it.

Figure 6.1: The PlanetLab network

6.2 Application-Level Clustering

This section presents the performance of measures with ALC. First, we are going to explain our approach (section 6.2.1). Second, we are going to discuss the measures.

6.2.1 Introduction To The Measure Scenario

The measures were performed on the Planet Lab network during the Easter holidays (from April 4th to 6th, 2003). The Planet Lab nodes used were the following:

Lancaster (`planetlab1.cs-ipv6.lancs.ac.uk` - 194.80.38.242)

Arizona (`planetlab1.cs.arizona.edu` - 150.135.62.2)

Cambridge (`planetlab1.xeno.cl.cam.ac.uk` - 128.232.103.201)

Copenhagen (`planetlab1.diku.dk` - 192.38.109.143)

University of Bologna (`planetlab1.cs.unibo.it` - 130.136.254.21)

University of Technology, Sydney (`planetlab1.it.uts.edu.au` - 138.25.15.194)

University of Chicago (`planetlab1.cs.uchicago.edu` - 128.135.11.149)

Kansas (`kupl1.ittc.ku.edu` - 129.237.123.250)

UC Santa Barbara (`planet1.cs.ucsb.edu` - 128.111.52.61)

The measure were performed in an environment containing no firewall and no NAT.

The Lancaster node was chosen to be the root of the hierarchy. The port used for the ALC application was 5000, except for Arizona and Santa Barbara where it was both 5000 and 7000. So, there were two ALC applications running on the Arizona and on the Santa Barbara nodes.

The ALC application was launched in this order: Lancaster, Arizona 1, Arizona 2, Cambridge, Copenhagen, Bologna, Sydney, Chicago, Kansas, Santa Barbara 1 and, finally, Santa Barbara 2.

The nodes measured were the following: Lancaster, Cambridge, Arizona 1, Santa Barbara and Kansas. The measures were done by taking "statistics" in the application all the minutes during approximately 15 - 20 minutes.

The only parameter was the value for the Probabilistic Join. It was put to 5. Remember that the Probabilistic Join represents the maximum number of nodes a new comer will consider at each step of the Join Procedure (see section 4.2).

The following points were measured: the number of heartbeat messages exchanged between nodes, the number of `OBJREQ`[1] messages exchanged, the number of Maintenance Procedure, the total number of messages exchanged, the total bytes exchanged between nodes and the time needed for a node to find its place in the hierarchy. These measures aim to show that our Java implementation of ALC works correctly.

6.2.2 Construction Of The Tree

Figure 6.2 shows the tree shape obtained during our test. The surrounded nodes are those that were the measure target. We could notice that the non-constrained aspect of the tree prevents to do speculation about its shape.

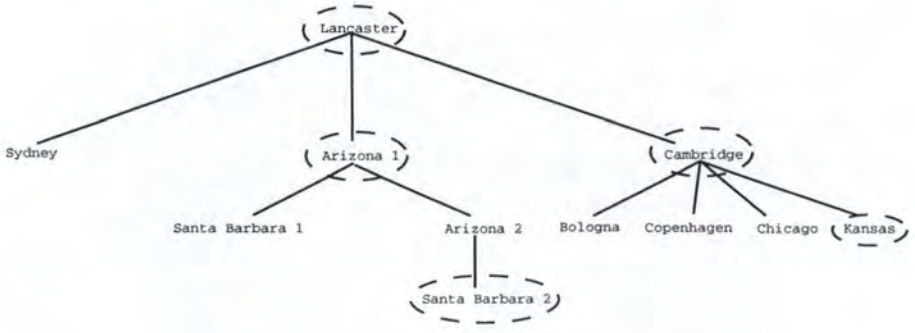


Figure 6.2: Tree shape

The tree shape does not seem obvious at the first look (Chicago is a child of Cambridge, Santa 2 is a child of Arizona 2, ...). To remove this misunderstanding, we are going to explain in details the complete Join Procedure.

First, we launch Lancaster as the root. Next, we launch Arizona 1 that contacts Lancaster with a `JOIN` message. As Arizona 1 is Lancaster's only child, Lancaster accepts immediately Arizona 1. Next, we launch Arizona 2 that contacts Lancaster with a `JOIN` message. Based on the measurement included in the `JOIN` message, Lancaster builds a region and checks if a child falls in it. As it is, Lancaster sends back a `TRY` message containing information about Arizona 1 and the radius information. Arizona 2 measures its distance to Arizona 1 and, according to the radius, contacts Arizona 1 as the new potential parent. Arizona 1 accepts Arizona 2 as its child. Next, we launch Cambridge. It contacts Lancaster with a `JOIN` message. Lancaster builds a region and constats that none child falls in it. Remember that the region is built with

[1]The different messages used by ALC are explained in appendix A

the measure performed by the new comer (the best measurement is put to zero and the worst is put to the RTT). In this case, the RTT between Cambridge and Lancaster is less than the RTT between Arizona 1 and Lancaster. So, Lancaster accepts Cambridge as its child.

We next launch Copenhagen that contacts Lancaster with a `JOIN` message. The region indicates to Lancaster that Cambridge could be a potential parent for Copenhagen. Thus, Lancaster sends back a `TRY` message containing informations about Cambridge and the radius information. Then, Copenhagen starts again a Join Procedure with Cambridge as potential parent because Cambridge is in the radius. Cambridge accepts Copenhagen as its child. Next, we launch Bologna. The mechanism is identical to the one used by Copenhague because no child falls into the region built by Cambridge (Bologna is closer to Cambridge than Copenhagen). We next launch Sydney. As Sydney is closer to Lancaster than to Cambridge and Arizona, it becomes a Lancaster's child. For Chicago and Kansas, the mechanism is identical to the one used by Bologna. Santa Barbara 1 is an Arizona 1's child because Arizona 1 felt in the region built by Lancaster, Arizona 1 felt in the radius and is closer to Santa Barbara 1 than Lancaster. Santa Barbara 2 is an Arizona 2's child because when Arizona 1 built the region, only Arizona 2 felt in it (Arizona 2 is closer to Arizona 1 than Santa Barbara 1).

The tree built is steady, as explained in section 6.2.7. A maintenance procedure is nevertheless useful because nodes can join/leave the tree dynamically. The common theme linking the next sections will be the measure of the maintenance cost.

6.2.3 Number Of Messages Exchanged

There are two kinds of messages exchanged in ALC: the messages used by the Join Procedure and the messages used for the tree maintenance. This section aims to show the total number of messages exchanged by the nodes.

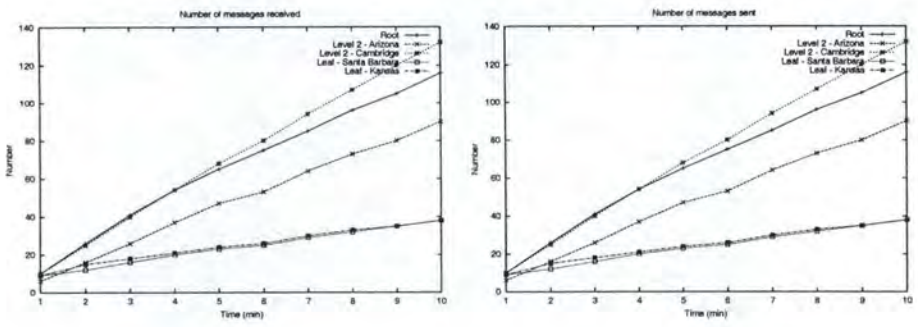
Figure 6.3 shows the total number of messages exchanged between nodes during ten minutes. As shown, the higher in the tree a node is and more it has children, more messages it has to handle (sent or receipt). The curves are cumulative. In average, after ten minutes, the Cambridge nodes received and sent 132 messages. In opposite, the Kansas node received and sent 38 messages. This shows the difference between a leaf and an important node in a load point of view.

Figure 6.3 also shows that Cambridge exchanges more messages than Lancaster. This is due to the fact that Cambridge has a more important neighborhood: a parent and four children. In opposite, Lancaster has "only" three children and no parent. Thus, the number of maintenance messages exchanged by Cambridge is more important. We can also note that, during the first four minutes, the number of messages exchanged by Lancaster and Cambridge is identical. The reason is simple: Lancaster is always the starting point for a new comer's Join Procedure. This is not necessarily the case for Cambridge, even if its probability to be involved in a Join Procedure is high.

As figures 6.3(a) and 6.3(b) suggest, the number of messages sent equals the number of messages received. Thus, a node sends a message in response of another or waits the answer of a message sent.

6.2.4 Heartbeat Messages Exchanged

Heartbeat messages are part of the maintenance messages. They are used to check if a neighbor (parent or child) is still alive.



(a) Total number of messages received (b) Total number of messages sent

Figure 6.3: Total number of messages exchanged

As shown in figure 6.4, the growth of the heartbeat messages exchanged is more or less linear during ten minutes. The heartbeat mechanism is lightweight for the leaves of the tree. It is easy to understand: a leaf exchanges heartbeat messages with only one node: its parent. In opposite, the mechanism is more heavy for nodes having a lot of children, as, for example, Cambridge (four children and a parent). More than 20 messages were exchanged during ten minutes. In average, a heartbeat message is sent or received each 30 seconds even if the node has received informations from a neighbor. This mechanism is lightweight for the network (heartbeats are small messages) but it represents a higher cost for the nodes at the state information and the CPU level. This problem could be solved easily: the timer for the heartbeat messages could be set up only when a node does not receive any messages from a node in its neighborhood during 30 seconds.

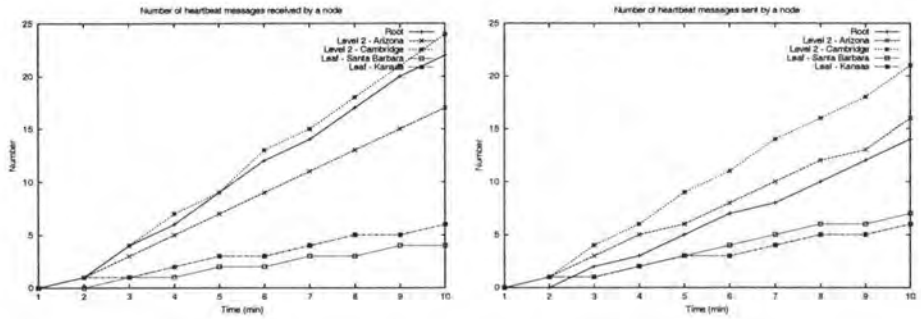
The curve order is a little bit different in figure 6.4(a) and 6.4(b). This could be explained by the timer used to schedule the heartbeat messages sending. This timer contains a random part to avoid collision and a node overloading.

In our implementation, the timer value negotiated by peers during the Join Procedure belongs to [45;125] seconds. Regarding the number of nodes in Cambridge's neighborhood, Cambridge, during ten minutes, should send between 24 and 65 heartbeat messages. With regard to a leaf, it should send between 5 and 13 heartbeat messages. The measures have shown that the number of heartbeat messages sent by a peer is closer to the minimum than the maximum.

6.2.5 `OBJREQ` Messages Exchanged

Figure 6.5 is about the exchange of `OBJREQ` messages. An `OBJREQ` message is a message requesting informations to another peer. In response, an `OBJRSP` message is sent. The `OBJREQ` message can be sent for three reasons:

1. the Join Procedure always begins by measuring the RTT to the potential parent. Each node owns a measurement address used to receive probes for RTT measuring. A joining node could know this address by sending an `OBJREQ` message requesting the measurement address to the potential parent.



(a) Number of heartbeat messages received

(b) Number of heartbeat messages sent

Figure 6.4: Number of heartbeat messages exchanged

2. a timer is associated to the measurement and data address. When the timer is out, an `OBJREQ` message is sent, requesting the measurement and data address of the node.

3. a timer is associated to the RTT measured for each node. When this timer expires a new RTT measure is performed. For implementation reasons, it is realized by sending first an `OBJREQ` message to check the measurement address of the measured node, even if this measurement address is still available in the cache.

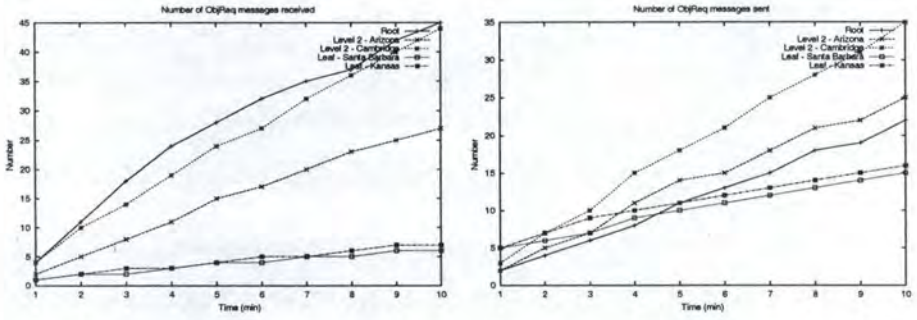
So, the `OBJREQ` message belongs to the messages used by the Join Procedure and by the Maintenance Procedure.

Figure 6.5(a) shows the number of `OBJREQ` messages received by a node during ten minutes. In opposite, figure 6.5(b) shows the number of `OBJREQ` messages sent by a node during ten minutes. We can see that a leaf sends more `OBJREQ` messages than it receives. Lancaster receives more messages than Cambridge, but it is the opposite for the sent.

The higher a node is in the hierarchy, the higher is the probability to be involved in the Join Procedure and, thus, the higher is the probability to receive `OBJREQ` messages. The more relationships a node has (as Cambridge, for example), the more addresses it has to manage. For example, let's take the case of the Cambridge node. This node has four children and one parent. Each of these nodes has a measurement address and a data address. A timer is associated to each address. So, Cambridge has to manage ten different addresses with ten timers. It has thus to send regularly ten `OBJREQ` messages to update its cache (figure 6.5(b). In the same way, each node in its neighborhood has to manage the Cambridge's measurement and data addresses. Thus, regularly, Cambridge receives `OBJREQ` messages requesting its measurement and data addresses (figure 6.5(a)). In opposite, a node lower in the hierarchy (Kansas, for example), sends and receives less `OBJREQ` messages because it has only one node's data and measurement addresses to manage.

The curves in figures 6.5(a) and 6.5(b) are different. This could be explained by the purpose of an `OBJREQ` message and the position of a node in the hierarchy. If the node is the root, it will receive a lot of `OBJREQ` messages because it is a mandatory path for the Join Procedure. Remember that the Join Procedure always begins by measuring the distance between the new comer and the potential parent. To achieve that, the new comer must know the potential

parent's measurement address and an OBJREQ message is sent to discover it. The root will send less OBJREQ messages because this node sends this message only to update its cache entry. Here, Lancaster has only three children, thus only three entry in its cache. A node like Cambridge will receive a lot of OBJREQ messages but fewer than the root. In opposite, it will send a lot of OBJREQ messages because, as said above, it has to manage a lot of entries in its cache (four children and one parent). If a node is a leaf (as Santa Barbara 2), it will send more OBJREQ messages than receive them because of the Join Procedure. If a node receives a TRY message containing a lot of potential parents, it has to measure its distance to all these nodes and thus, send first OBJREQ messages to know their measurement address. A leaf will receive very little OBJREQ messages because it is in relation with only one node: its parent.



(a) Number of OBJREQ messages received

(b) Number of OBJREQ messages sent

Figure 6.5: Number of OBJREQ messages exchanged

6.2.6 Total Bytes Exchanged

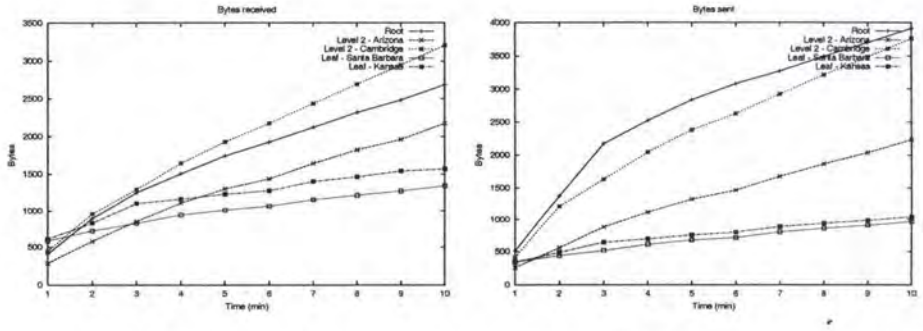
This section will evaluate ALC in terms of bytes exchanged between nodes.

After the first minutes, the total bytes exchanged between nodes is higher for the leaf (see figure 6.6). This could be explained by the Join Procedure, and more precisely the sending of JOIN messages and the receipt of TRY messages. The lower in the tree a node is, more it sends JOIN messages and more it receives TRY messages. A JOIN message contains two different addresses: the root address and the potential parent address. Theses addresses are present for security reasons but it makes messages longer. The root address will be used by the potential parent to check if the new comer does not attempt to join the wrong tree. If the root address does not correspond to the current root address of the tree, the potential parent sends back an ERROR message. The potential parent address will be used by the potential parent to check if the new comer contacts the right parent. If not, it sends back an ERROR message. A TRY message contains the addresses (signalling, measurement and data addresses) of a node's children. More a node has children, more heavy is the TRY message. It is the case, for example, for the Cambridge node. After three minutes, it seems to stabilize. The growth becomes linear. This is due to the tree stability (see further).

As shown in figure 6.6(a), the load is the most important for Cambridge. Again, it is due to the messages exchanged during the Join Procedure. Add to this the fact that this node

receives more messages (see figures 6.3(a) and 6.3(b)). All these elements put together, it leads to a load increase of nodes having an important neighborhood. The depth of the tree has also an impact on the load. The probality for a node in a high level in the tree to be a mandatory route during a Join Procedure is higher than for a node in a low level in the tree. Furthermore, the number of Join Procedure needed to find a place in the hierarchy grows with the depth of the tree.

As shown in figure 6.6(b), the load is the most important for Lancaster (the root). The growth of the load is important during the three first minutes. Again, this could be explained by the Join Procedure, and particulary by the sending of `TRY` messages. After the three first minutes, the growth becomes linear.



(a) Bytes received (b) Bytes sent

Figure 6.6: Total length exchanged

All these measures about messages confirm a theorical problem: the non-constrained aspect of the tree can lead to an overloading of a node. This problem is not desirable for the lightness of the protocol.

6.2.7 Tree Maintenance

As said several times, a node can join/leave the tree dynamically. This could have an impact on the structure of the tree and, after a while, the current position of a node may not be the best one.

This section will discuss the number of Maintenance Procedures performed by a node during ten minutes. We can note that no intentional departure of a node[2] was realized during these ten minutes.

In average, we could see that a node performs a Maintenance Procedure each minute. Figure 6.7 shows the number of Maintenance Procedures that does not result in a move of the node. However, the spacing of Maintenance Procedures due to the successive number of procedure without a move is not shown with force. After ten minutes, each node has performed nine Maintenance Procedures. This could be explained by the choice for the value used by the maintenance timer. As said in section 4.5, the maintenance timer is given by $\min[B(1+\delta)^i, M]]$,

[2]We mean by *intentional departure* the fact that the User component orders the Agent component to leave the tree by sending a *leave* event (see section 5.2.1.3.

where B is the minimum maintenance interval, M is the maximum maintenance interval, δ is a scaling factor ($\delta > 0$) and i is the number of previous and consecutive Maintenance Procedures that didn't result in a move of a node. For our Java implementation, B was put to 30, M to 60 and δ to 0.1. The first Maintenance Procedure should be performed quickly to make sure the node is well positioned in the tree. At worst, a Maintenance Procedure will be performed each minute. This proximity of maintenance procedure is desirable to make sure the position is still the best one in a dynamic environment.

During the performance measurement, we noticed the stability of the tree. Only two nodes (Kansas and Santa Barbara 2[3]) performed one maintenance procedure that result in a move. It was performed quickly (the first or the second maintenance procedure execution). This stability could be explained by two factors:

1. the RTT stability. We mean by stability the fact that the way we implemented the RTT measurement gives us nearly the same RTT between two nodes, with a variation of two milliseconds, on condition that the underlying network topology does not change between two measurements.

2. the modification of a Join Procedure aspect. A node will go down one layer if and only if there is at least a node in the joining node scope (i.e. within the radius) and the distance towards this node is less than the distance towards the potential parent. This aspect is shown by figure 6.8. A is the root of the tree and N is the new comer. N builds a region with the radius given by A and measures its distance to D. As shown in the figure, the distance between N and A (m) is less than the distance between N and D (n). So, it is preferable for N to be a child of A than a child of D.

This stability is highly desirable as far as the tree has to transport data. Too much movements due to Maintenance Procedure would lead to instability harmful to data transporting.

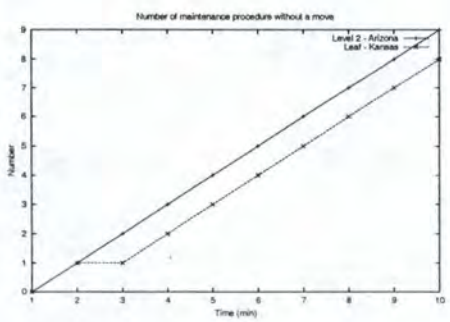


Figure 6.7: Number of maintenance procedure without a move

6.2.8 Time Needed To Find A Place In The Hierarchy

The time needed for a node to find its place in the hierarchy is a way to measure the Join Procedure latency (see figure 6.9).

[3]For readability reasons, the figure 6.7 shows only the Kansas and Arizona curves. The Kansas and Santa Barbara curves are identical and all the others curves are identical to the Arizona one.

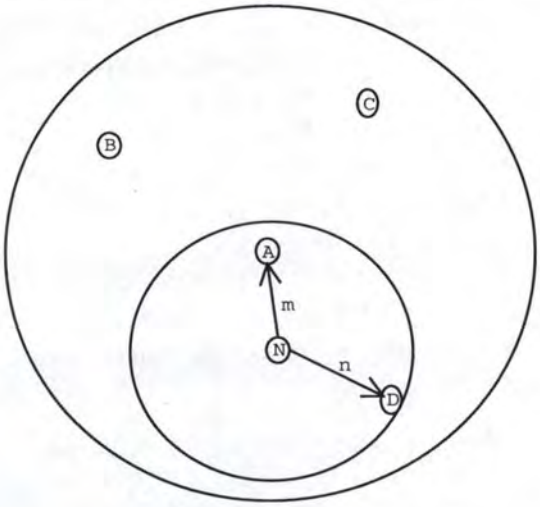


Figure 6.8: Modification of a Join Procedure aspect

The lower a node is in the hierarchy, the more time it needs to find its place in the hierarchy. This is principally explained by the time needed to evaluate the RTT and the fact that the node must evaluate more potential parents. We chose to use UDP to ping nodes. UDP is a non-reliable protocol. To make sure the pings are realized, we "protected" the RTT measurement with a mechanism ensuring that after one second, if the RTT is not measured, the pings are to be started again. Sometimes, it is necessary to launch two or three times the measurement. And when a measurement has to be done for each node in a `TRY` message, it could take a lot of time. This is why the time needed for Kansas to find its place is the highest. Kansas was situated at level three and was the 9th node launched.

The peak observed for Kansas can be interpreted as the worst case to find a place in the hierarchy: a lot of nodes to test, all the RTT measures need several pings, ...

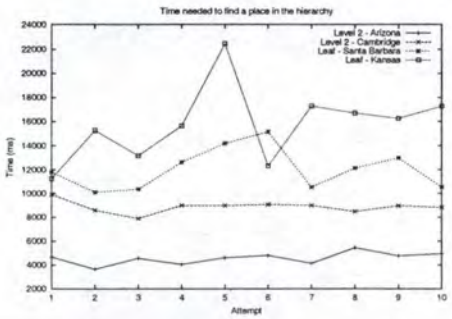


Figure 6.9: Time needed for a node to find its place in the hierarchy

Table 6.1 gives the average time to find a place in the hierarchy by level. The level 1 is set to zero because it corresponds to the root of the tree. Level 2 corresponds to Sydney, Arizona 1 and Cambridge nodes. Level 3 corresponds to Santa Barbara 1, Arizona2, Bologna, Copenhagen, Chicago and Kansas nodes. Finally, level 4 corresponds to Santa Barbara 2

node. This table allows us to see that the time needed to find a place in the hierarchy grows with the depth of the tree, as suggested by figure 6.9

Level in the tree	Average time to find a place (ms)
Level 1	0
Level 2	6777
Level 3	8455
Level 4	12043

Table 6.1: Average time to find a place by level

6.3 Tree Building Control Protocol

The measures were performed on the Planet Lab network between April 28th, 2003 and May 03rd, 2003. The nodes used were the same than the one used for ALC. The root, the port used and the start order were also identical. The only changing parameter was the fanout. It was put to three for all the nodes. Remember that the fanout represents the maximum number of children a node accepts to accommodate with (see section 4.3).

Some performance tests have been processed with TBCP. Unfortunately, because of lack of time, the tree stability can't be reached. The table 6.2 shows the number of Maintenance Procedures processed by some nodes during a period of ten minutes. As we can see, there are a lot of Maintenance Procedures resulting in a move of a node. This aspect is clearly prejudicial for TBCP and can be explained by the fact that the Maintenance Procedure (nearly identical to the one used by ALC - see pseudo-code 2) is not adapted to TBCP. Because a node does not choose its place in the tree, it cannot take the initiative of a Maintenance Procedure. We hope that this problem will be fixed in further version of TBCP.

Following the example of ALC, the inadequacy of the Maintenance Procedure can be solved by introducing a `TENTATIVE` object in messages exchanged during the Join Procedure. This object should indicate to the potential parent that the new comer does not want to join immediately the tree but wants to detect the effects of joining. The `TENTATIVE` object should be added in the `HELLO`, `JOIN` and `WELCOME` messages. This object should also be optional. The new comer will effectively join the tree if the new place is better than the previous one, i.e. higher in the tree. This is performed by sending a `WELCOMEACK` message in response of the `WELCOME` message with the `TENTATIVE` object.

Node	Maintenance without a move	Maintenance with a move
Cambridge	3	6
Chicago	3	4
Bologna	8	1
Copenhagen	8	1
Santa Barbara 2	3	4
Sydney	1	8

Table 6.2: Tree instability

However, the test draft showed some interesting aspects of TBCP: the time needed to find a place and the tree shape. First, we noted that the time a node needs to find a place in the tree is higher than in ALC. This time varies between 5000 ms (for a node in a high level) and 105000 ms (for a node in a low level in a very extreme case). This range can be explained by the fact that a potential parent can process only one Join Procedure at a time. So, if two nodes want to join the same parent at the same time, one will have to wait during 15 seconds plus a random time between zero and ten. This random time is used to avoid collision. Second, we noticed during debugging session that the tree shape is always the same, independently of the starting order. This is due to the score function and the RTT stability. Thus, if we know the distance between nodes, we can do speculations about the tree shape.

6.4 Conclusion

The measures performed on ALC showed that our implementation works: the tree is built correctly and is steady. The measures brought also a potential node overloading problem to the light. This problem could be solved with the `LEAFONLY` object[4]. This object indicates that the node does not wish to have children. It could highly limit the load of nodes that do not have the needed resources (CPU, memory) allowing to deal with.

The measures performed on TBCP were not complete because of lack of time. However, several interesting results were observed: the tree construction is carried out correctly and it exists speculation possibilities about the tree shape. Nevertheless, some negative results were also noted: the inadequacy of the Maintenance Procedure and the time needed to find a place in the tree. For the Maintenance Procedure, an idea of the solution was proposed.

[4]see appendix A.2.3.7.

Chapter 7

Conclusion

This dissertation is coming to an end. The limitations of IPv4 were presented and a solution to solve them (IPv6) was explained.

The motivations for multicast were discussed and the protocols used to implement multicast at layer three were introduced. The drawbacks of these protocols were presented and a solution to solved them, Application Level Multicast, was developed through overlays.

We presented advantages of overlays: they are incrementally deployable, adaptable, robust, customizable and standardized. We also discussed their drawbacks: management complexity, less efficient that code running in routers, firewalls, NAT, proxies, effort needed to deduce the network topology.

We described several well known overlays and summarized their important points in a table.

ALC and TBCP were completely described: their functioning (i.e. how they build a logical tree) was explained and our fully implementable specification was proposed in appendix. This specification includes messages format definition, ABNF, internal node behavior and complete finite state machine. A comparison between ALC and TBCP was proposed and discussed three aspects: the philosophy of the tree built, the acceptance level of a new comer in the tree and the way of accepting a new comer. The theorical limitations of both overlays were shown and will be discussed in section 7.1. We extended to ALC and TBCP the table summarizing the important points of overlays.

Our Java implementation was presented by describing the cutting of each overlay in four logical components. We also introduced the Java code by giving a short explanation of each Java package, classe and interface. We proposed a manner to transfer data over the tree built by ALC and TBCP. Our idea is the following: the application above ALC and TBCP (i.e. the application using them) has to manage the data transfer. The interactions between the application and the underlying overlay protocol will be possible by using the User component. We described the events that should be added between the User component and the Agent/Controller component.

The performance measures performed on ALC showed that our Java implementation of the overlay is working correctly. That means that the protocol is respected (i.e. the tree is built correctly) and the tree built is steady. The measures brought also a potential node overloading problem to the light. This problem could be solved with the `LEAFONLY` object.

The performance measures performed on TBCP are not complete because of lack of time. However, several interesting results have been observed: the tree construction is carried out

correctly and it exists speculation possibilities about the tree shape. Some negative results were also noted: the inadequacy of the Maintenance Procedure and the time needed to find a place too high.

7.1 Further Works

However, the work is not yet finished. We have seen that ALC and TBCP suffered from several limitations: root not fault tolerant, NAT/firewalls problems, root path updating, ... We wish these problems will be fixed in further versions of both overlays.

For the fault tolerant problem, we proposed an idea of solution: the root replication. As for Overcast, we could imagine that the three first levels of the hierarchy have degree one and these nodes are a replication of the root. If the root dies, the node in the next level becomes the root. This solution will be easy to implement in TBCP by simply fixing the fanout to one for the three first levels. For ALC, the non-constrained aspect of the tree makes this solution more difficult to implement. A new type of object (`ROOTREPLICATION`) should be defined.

For the root path updating, a procedure should be defined to propagate changes in the tree, as the Up/Down protocol in Overcast [5].

The performance measures performed on TBCP showed that the tree built is not steady. This is brought about the inadequacy of the Maintenance Procedure. We also proposed an idea of solution. Following the example of ALC, we could introduce a `TENTATIVE` object in some messages exchanged during the Join Procedure in TBCP. This object should indicate to the potential parent that the new comer does not want to join immediately the tree but wants to detect the effects of joining. The new comer will effectively join the tree if the new place is better than the previous one, i.e. higher in the tree.

Despite of all the efforts deployed, ALM is not yet set as the standard for multicast. It only remains to see if ALM will achieve to get the upper hand.

Bibliography

[1] S. E. Deering. Hosts Extensions For IP Multicasting, RFC 988. *IETF*, Jul. 1986.

[2] S. E. Deering. Host Extensions For IP Multicasting, RFC 1054. *IETF*, May 1998.

[3] R. Canonico P. Smith, L. Mathy and D. Hutchison. ALM and ProgNets for v4-to-v6 Multicast Transition. In *IEEE OpenArch'01 Short Paper Session*, Anchorage, Alaska, USA, Apr. 2001.

[4] Web site about Application Level Overlays: `http://www.activenet.lancs.ac.uk/overlay/`.

[5] J. Jannotti D. K. Gifford K. L. Johnson M. F. Kasshoek and J. W. O'Toole Jr. Overcast: Reliable Multicasting with an Overlay Network. In *USENIX OSDI 2000*, San Diego, CA, USA, Oct. 2000.

[6] S. Simpson L. Mathy, Y. Wakeman and D. Hutchison. Universal, Efficient and Scalable Neighbour Discovery. Lancaster University, Internal Report, 2002.

[7] R. Canonico L. Mathy and D. Hutchison. An Overlay Tree Building Control Protocol. In *Proc. of 3rd Intl. COST264 Workshop on Networked Group Communication (NGC 2001)*, Nov. 2001.

[8] J. Postel. Internet Protocol, RFC 791. *IETF*, Sept. 1981.

[9] Etude Réalisée pour la DIGITIP par l'Idate. Les enjeux du Déploiement du Protocole IPv6 - Rapport Final. `http://www.telecom.gouv.fr/documents/etu\_accueil.htm`, 2002.

[10] R. Atkinson S. Kent. IP Encapsulating Security Payload (ESP), RFC 2406. *IETF*, Nov. 1998.

[11] S. Thomson and T. Narten. IPv6 Stateless Address Autoconfiguration, RFC 2462. *IETF*, Dec. 1998.

[12] H. Sandick B. Haberman and G. Kump. Protocol Independent Multicast Routing in the Internet Protocol Version 6 (IPv6). *Internet Draft, pim-ipv6-03.txt, Work In Progress*, Mar. 2000.

[13] S. Bhattacharyya C. Diot L. Giuliano R. Rockell J. Meylor D. Meyer G. Shepherd and B. Haberman. An Overview of Source-Specific Multicast (SSM) Deployment. *Internet Draft, draft-ietf-ssm-overview-02.txt, Work In Progress*, Dec. 2001.

[14] H. Holbrook and B. Cain. Source-Specific Multicast for IP. *Internet Draft, draft-ietf-ssm-arch-02.txt, Work In Progress*, Mar. 2003.

[15] C. E. Perkins and D. B. Johnson. Mobility Support in IPv6. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, Rye, New-York, USA, Nov. 1996.

[16] R. Gilligan and E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers, RFC 1933. *IETF*, Apr. 1996.

[17] S. Deering D. Estrin D. Farinacci V. Jacobson A. Helmy D. Meyer and L. Wei. Protocol Independant Multicast Version 2 Dense Mode Specification. *Internet Draft, draft-ietf-pim-v2-dm-03.txt, Work In Progress*, Jun. 1999.

[18] D. Estrin D. Farinacci A. Helmi D. Thaler S. Deering M. Handley V. Jacobson C. Liu P. Sharma and L. Wei. Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification, RFC 2362. *IETF*, Jun. 1998.

[19] W. Fenner S. Deering and B. Haberman. Multicast Listener Discovery (MLD) for IPv6. *IETF*, Oct. 1999.

[20] B. Haberman and R. Worzella. IP Version 6 Management Information Base for The Multicast Listener Discovery Protocol. *IETF*, Jan. 2001.

[21] C. Partridge D. Waitzman and S. Deering. Distance Vector Multicast Routing Protocol, RFC 1075. *IETF*, Nov. 1988.

[22] D. Katz T. Bates, R. Chandra and Y. Rekhter. Multiprotocol Extensions for BGP-4, RFC 2283. *IETF*, Feb. 1998.

[23] D. Meyer and B. Fenner. Multicast Source Discovery Protocol (MSDP). *Internet Draft, draft-ietf-msdp-spec-14.txt, Work In Progress*, Nov. 2002.

[24] J. Moy. Multicast Extensions for OSPF, RFC 1584. *IETF*, Mar. 1994.

[25] CISCO Systems. IP Multicast Training Materials. ftp://ftpeng.cisco.com/ipmulticast/training/index.html, Aug. 2001.

[26] Yunxi Sherlia Shi. *Design Of Overlay Networks For Internet Multicast.* PhD thesis, Sever Institute of Washington University, Aug. 2002.

[27] Ian Stoica. CS268: Lecture 19 (Application Level Multicast. `http://www.cs.berkeley.edu/~istoica/cs268/notes/lecture19.pdf`, Mar. 2001.

[28] S. G. Rao Y. Chu and H. Zhang. A Case for End System Multicast. In *ACM SIGMETRICS 2000*, Santa Clare, CA, USA, June 2000.

[29] D. Verma D. Pendarakis, S Shi and M. Waldvogel. ALMI: An Application Level Multicast Infrastructure. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 49–60, San Francisco, CA, USA, Mar. 2001.

[30] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, Nov. 2001.

[31] P. Francis. Yoid: Extending the Internet Multicast Architecture. `http://www.icir.org/yoid/docs/index.html`, Apr. 2000.

[32] A.-M. Kermarrec M. Castro, P. Druschel and A. Rowstron. Scribe: A Large-Scale and Decentralized Application-Level Multicast Infrastructure. In *IEEE Journal on Selected Areas in Communications*, volume 20, NO 8, Oct. 2002.

[33] F. Kaashoek D. Andersen, H. Balakrishnan and R. Morris. Resilient Overlay Networks. In *18th ACM Symposium on Operating System Principles (SOSP)*, Banff, Canada, Oct. 2001.

[34] Q. H. Mahmoud. Network Programming with JavaTM 2 Platform, Standard Edition 1.4 (J2SETM). `http://java.sun.com/features/2002/08/j2se-network.html`, Sept. 2002.

[35] M. T. Nygard. Master Merlin's New I/O Classes. Squeeze Maximum Performance Out of Non-blocking I/O and Memory-mapped Buffer. `http://www.javaworld.com/javaworld/jw-09-2001/jw-0907-merlin\_p.html`, Sept. 2001.

[36] T. Burns. Non-blocking Socket I/O in JDK 1.4. http://www.owlmountain.com/tutorials/NonBlockingIo.htm, Dec. 2001.

[37] J. Zukowski. New I/O Functionality for JAVA TM 2 Standard Edition 1.4. `http://developer.java.sun.com/developer/technicalArticles/releases/nio/%`, Dec. 2001.

[38] PlanetLab web site: `http://www.planet-lab.org`.

[39] D. Culler L. Peterson, T. Anderson and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the First ACM Workshop on Hot Topics in Networks (HotNets - I)*, Princeton, NJ, Oct. 2002.

[40] L. Peterson and T. Roscoe. PlanetLab Phase 1: Transition to an Isolation Kernel. `http://www.planet-lab.org`, Sept. 2002.

[41] L. Peterson. Dynamic Slice Creation, The PlanetLab Architecture Team. `http://www.planet-lab.org`, Work In Progress, Oct. 2002.

[42] Abilene web site: `http://abilene.internet2.edu`.

[43] Internet2 web site: `http://www.internet2.edu/about/aboutinternet2.html`.

[44] D. Crocker and P. Overell. Augmented BNF for Syntax Specifications : ABNF, RFC 2234. *IETF*, Nov. 1997.

[45] R. Hinden and S. Deering. IP Version 6 Addressing Architecture, RFC 2373. *IETF*, Jul. 1998.

[46] R. Hinden and S. Deering. Internet Protocol Version 6 Specification, RFC 2460. *IETF*, Dec. 1998.

[47] O. Bonaventure. Info2210 - Téléinformatique et Réseaux: Fonctions et Concepts. FUNDP - Institut d'Informatique. Lessons, Academic Year 2001 – 2002.

[48] JavaDoc: `http://www.java.sun.com`.

Appendix A

ALC: Implementation Document

This appendix represents the document used to implement the ALC protocol.

A.1 Terminology

This section contains the definitions of concepts/terms used in this chapter.

- *Node*: a node is an end-host and/or tansport/application-level proxies or servers.
- *Agent*: an agent is the part of a node that participates in building a cluster hierarchy. It is also called *Cluster Agent*
- *Message*: a message has a type and contains zero or more objects. A Cluster Agent reacts according to the type of the message.
- *Object*: an object has a type and contains data associated with that type. Objects are component of messages.
- *Peer*: participant of the application-level overlay network.
- *Cluster*: a cluster is represented by a cluster head and is composed of the cluster head and other nodes.

A.2 Protocol

This section describes the network interactions between peers in a clustering hierarchy. This includes the messages, object types, and their formats. It does not describe how they are transported from a peer to another.

Internal architecture and behavior of a node is beyond the scope of this section. Section 5.1 suggested an internal architecture. The behavior for a peer is described in appendix A.4.

A.2.1 Lexical Elements

We define here lexical elements that will be used in the rest of this appendix. The Augmented BNF (ABNF) [44] is used for the syntax specification.

bit ::= "0" | "1"

Octet ::= 8*8bit

time ::= 15*bit

objlength ::= 2*2Octet *; object length is stored on 2 bytes.*

msglength ::= 4*4Octet *; message length is stored on 4 bytes.*

Port ::= 2*2Octet

IPv6_address ::= 16*16Octet

address ::= IPv6_address Port *; an IPv6 address (including v4-mapped) and a port number.*

A.2.2 Basic Types

Here are some more complex types. They use the lexical elements defined above in section A.2.1.

appdata ::= *Octet *; a sequence of octets with application-specific meaning.*

timed-appdata ::= appdata time bit *; appdata plus an expiry time, and a boolean indicating whether the data can be passed on to another peer.*

sigaddr ::= 1*2address *; a signalling address – an IPv6 address and port number, and/or an IPv4 mapped IPv6 address and port number. It identifies a peer in an application-level overlay and is sufficient for signalling with that peer.*

measurement ::= appdata *;an application-specific measurement.*

A.2.3 Object Types

Objects are components of messages. The object type specifies the type of the data it contains, and broadly indicates the purpose of the object. Objects with unrecognized types should be ignored.

A.2.3.1 DADDR Object

This object consists of a **timed-appdata**, and represents the data address of a peer.

DADDR_body ::= timed-appdata

A.2.3.2 MADDR Object

This object consists of a **timed-appdata**, and represents the measurement address of a peer.

MADDR_body ::= timed-appdata

A.2.3.3 PEERADDR Object

This object consists of an **address**, and represents the signalling address of a parent. A second **address** holds an alternative address for dual v4/v6-capable hosts. The **address** could be an IPv4 address only, an IPv6 address only or an IPv4 and IPv6 address.

PEERADDR_body ::= 1*2address

A.2.3.4 ROOT Object

This object consists of an **address**, and represents the signalling address of the root of the tree. A second **address** holds an alternative address for dual v4/v6-capable hosts.

ROOT_body ::= 1*2address

A.2.3.5 KEY Object

This object consists of an object identifier, and indicates an object being requested.

KEY_body ::= Obj_id *; Obj_id will be defined in section A.2.5*;

A.2.3.6 MEASUREMENT Object

This object consists of an **appdata**, and represents a measurement.

MEASUREMENT_body ::= appdata

A.2.3.7 LEAFONLY Object

This object contains no data. Its presence indicates a wish to not be the parent of any other node.

A.2.3.8 TENTATIVE Object

This object contains no data. Its presence indicates a wish to not join a parent, but to detect the effect of joining. The peer (a new comer) using this object wants to know its position in the hierarchy but without joining effectively the hierarchy.

A.2.3.9 TIMER Object

This object consists of a **time**, and represents the minimum timer value for the heartbeat message

TIMER_body ::= time

A.2.4 Message Types

Objects of unknown types, or objects unexpected within a particular type of message, will be ignored.

A.2.4.1 OBJREQ Message

This message is sent to a peer to request objects (for example DADDR, MADDR, ...), and contains the following objects:

keys a sequence of KEYs.

OBJREQ_body ::= *KEY_obj

A.2.4.2 OBJRSP Message

This message is sent as a response to an OBJREQ message, and contains the following objects:

objects a sequence of objects requested by a peer.

OBJRSP_body ::= *Object

A.2.4.3 JOIN Message

This message marks the start of a Join Procedure. It is sent by a node which wants to join the hierarchy to a potential parent, and contains the following objects:

root a ROOT indicating the root of the tree

parent an optional PEERADDR indicating the expected parent of the sender.

dist a MEASUREMENT made by the sender against the receiver.

tentative an optional TENTATIVE indicating that the sender is merely checking the outcome of attempting to join the receiver.

leafonly an optional LEAFONLY indicating if the sender will accept new clusters of its own. If not present, the sender will accept new clusters.

timer a TIMER indicating the minimum timer value for the heartbeat message sending from the receiver of the JOIN message to the sender

JOIN_body ::= 3*6Object

A.2.4.4 TRY Message

This message is sent by a potential parent to the new comer in response of a JOIN message only if children of this potential parent may be closer parent to the new comer, and contains the following objects:

radius a MEASUREMENT to help the receiver determine which of the *sibs* are suitable parents.

sibs a sequence of PEERADDRs, DADDRs and MADDRs. Each DADDR and MADDR is associated with the preceding PEERADDR.

TRY_body ::= 1*Object

A.2.4.5 NC Message

This message is sent by a new comer after receiving a TRY message if no child of the potential parent is a closer parent, and contains the following objects:

parent an optional PEERADDR indicating the expected parent of the sender.

dist a MEASUREMENT made by the sender against the receiver.

leafonly an optional LEAFONLY indicating that the sender will accept new clusters of its own.

timer a TIMER indicating the minimum timer value for the heartbeat message from the receiver of the NC message to the sender of this NC message.

NC_body ::= 2*4Object

A.2.4.6 NCA Message

This message is sent by a parent to welcome a new comer as its child, and contains the following objects:

objects a sequence of objects (i.e. DADDR, MADDR, ...) that may be useful in communicating arbitrary application data between the peers.

tentative an optional TENTATIVE indicating that the sender has not actually registered the receiver as its child.

timer a TIMER indicating the minimum timer value for the heartbeat message from the receiver of the NCA message to the sender

NCA_body ::= *Object

A.2.4.7 LEAVE Message

This message contains no objects, and is sent by a peer which wants to leave the hierarchy.

A.2.4.8 ERROR Message

This message contains no objects, and is sent to express that an error occurs.

A.2.4.9 ALIVE Message

This message contains no objects, and is sent to a peer to test if it is still alive.

A.2.4.10 ALIVEACK Message

This message contains no objects, and is sent to acknowledge an ALIVE message (and, broadly, to confirm that this peer is still alive).

A.2.5 Object Representation

In general, each object is transmitted as a sequence of octets, beginning with a length (the maximum length is 2^{16}), an object-type identifier and a body (the length accounts for all three parts).

Object ::= objlength Obj_id *Octet

The identifiers for each type are listed below:

Obj_id ::= DADDR_id | MADDR_id | PEERADDR_id | KEY_id | MEASUREMENT_id | LEAFONLY_id | TENTATIVE_id | ROOT_id | TIMER_id

DADDR_id ::= 0 *; identifier of the DADDR object*

MADDR_id ::= 1 *; identifier of the MADDR object*

PEERADDR_id ::= 2 *; identifier of the PEERADDR object*

KEY_id ::= 3 *; identifier of the KEY object*

MEASUREMENT_id ::= 4 *; identifier of the MEASUREMENT object*

LEAFONLY_id ::= 5 *; identifier of the LEAFONLY object*

TENTATIVE_id ::= 6 *; identifier of the TENTATIVE object*

ROOT_id ::= 7 *; identifier of the ROOT object*

TIMER_id ::= 8 *; identifier of the TIMER object*

An object of a particular type should have a body as defined in section A.2.3:

DADDR_obj ::= objlength DADDR_id DADDR_body

MADDR_obj ::= objlength MADDR_id MADDR_body

PEERADDR_obj ::= objlength PEERADDR_id PEERADDR_body

KEY_obj ::= objlength KEY_id KEY_body

MEASUREMENT_obj ::= objlength MEASUREMENT_id MEASUREMENT_body

LEAFONLY_obj ::= objlength LEAFONLY_id

TENTATIVE_obj ::= objlength TENTATIVE_id

ROOT_obj ::= objlength ROOT_id ROOT_body

TIMER_obj ::= objlength TIMER_id TIMER_body

Object ::= DADDR_obj | MADDR_obj | PEERADDR_obj | KEY_obj | MEASUREMENT_obj | LEAFONLY_obj | TENTATIVE_obj | ROOT_obj | TIMER_obj

A.2.6 Message Representation

In general, each message is transmitted as a sequence of octets, beginning with a length (the maximum length is 2^{32}), a message-type identifier and a body (a sequence of objects). The length accounts for all three parts.

Message ::= msglength Msg_id *Object

The identifiers for each type are listed below:

Msg_id ::= OBJREQ_id | OBJRSP_id | JOIN_id | TRY_id | NC_id | LEAVE_id | ERROR_id | ALIVE_id | ALIVEACK_id

OBJREQ_id ::= 0 *; identifier of the OBJREQ message*

OBJRSP_id ::= 1 *; identifier of the OBJRSP message*

JOIN_id ::= 2 *; identifier of the JOIN message*

TRY_id ::= 3 *; identifier of the TRY message*

NC_id ::= 4 *; identifier of the NC message*

NCA_id ::= 5 *; identifier of the NCA message*

LEAVE_id ::= 6 *; identifier of the LEAVE message*

ERROR_id ::= 7 *; identifier of the ERROR message*

ALIVE_id ::= 8 *; identifier of the ALIVE message*

ALIVEACK_id ::= 9 *; identifier of the ALIVEACK message*

A message of a particular type should have a body as defined in sect. A.2.4:

OBJREQ_msg ::= msglength OBJREQ_id OBJREQ_body

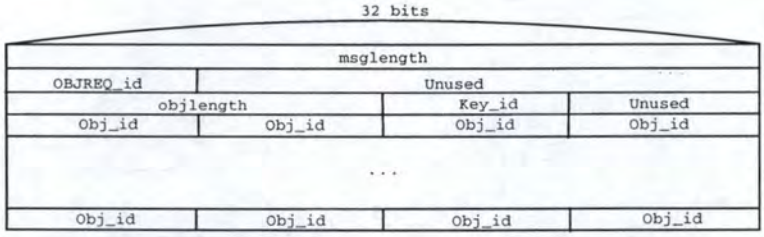


Figure A.1: The OBJREQ message

OBJRSP_msg ::= msglength OBJRSP_id OBJRSP_body

JOIN_msg ::= msglength JOIN_id JOIN_body

TRY_msg ::= msglength TRY_id TRY_body

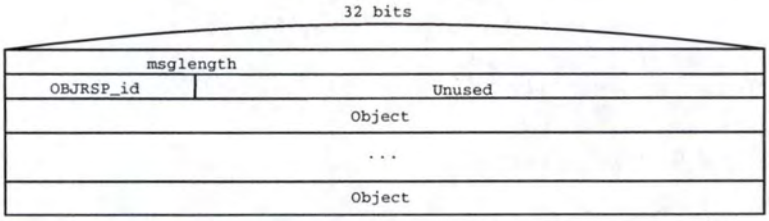


Figure A.2: The OBJRSP message

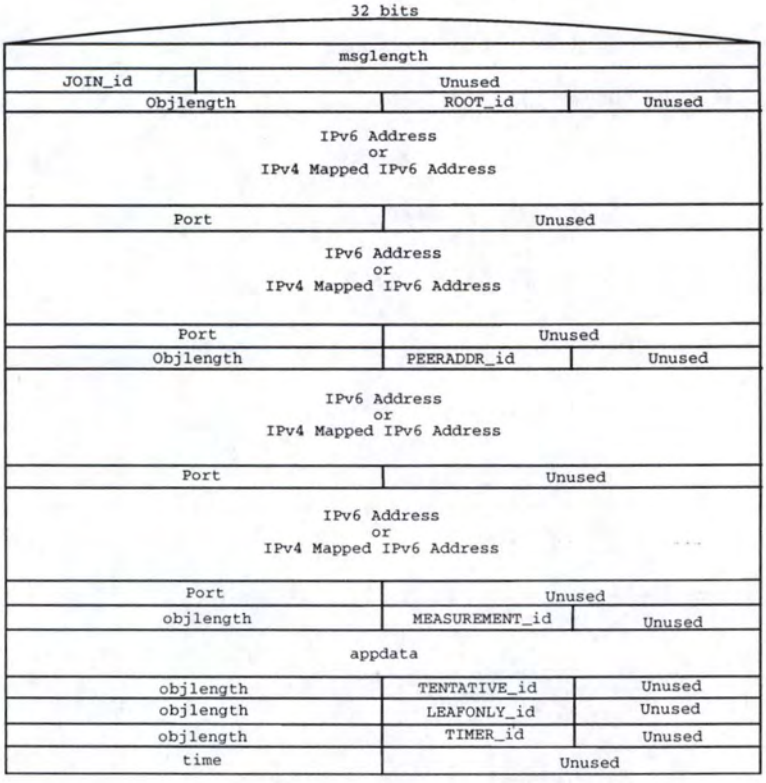


Figure A.3: The JOIN message

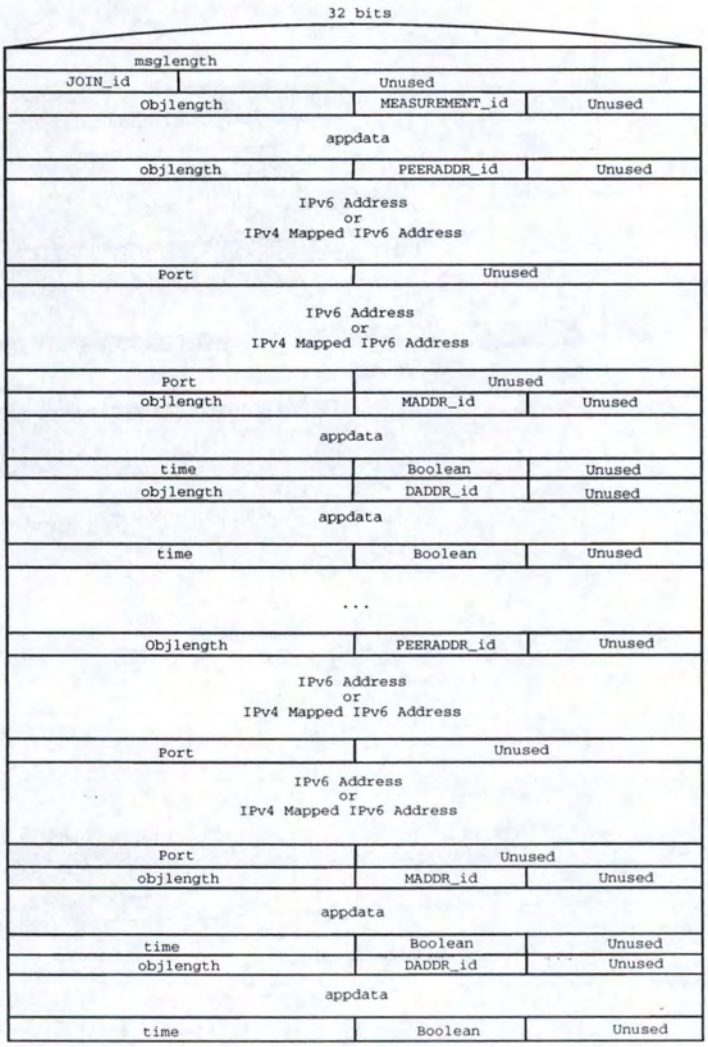


Figure A.4: The TRY message

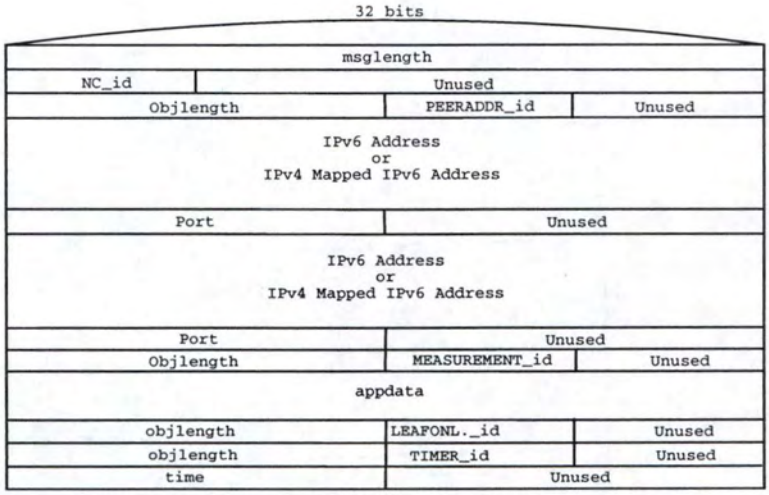


Figure A.5: The NC message

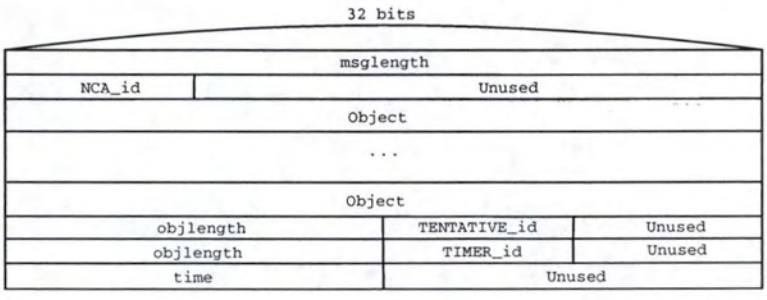


Figure A.6: The NCA message

NC_msg ::= msglength NC_id NC_body

NCA_msg ::= msglength NCA_id NCA_body

LEAVE_msg ::= msglength LEAVE_id

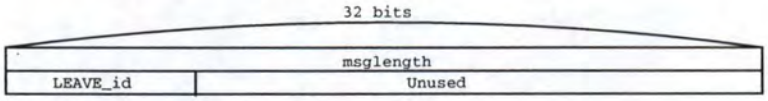


Figure A.7: The LEAVE message

ERROR_msg ::= msglength ERROR_id

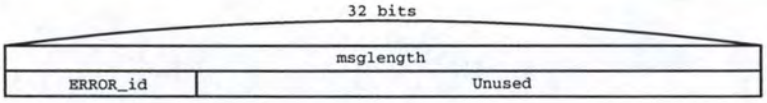


Figure A.8: The ERROR message

ALIVE_msg ::= msglength ALIVE_id

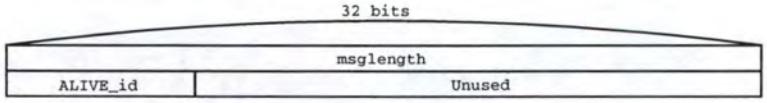


Figure A.9: The ERROR message

ALIVEACK_msg ::= msglength ALIVEACK_id

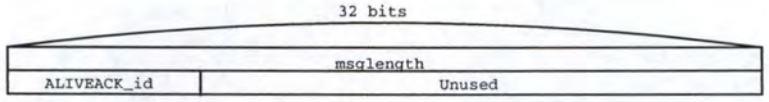


Figure A.10: The ALIVEACK message

A.3 State

An Agent retains information about its position in the hierarchy. Some of this (*the Agent state*) applies to its entire participation in the hierarchy, while the rest (*the peer state*) applies only to individual peers with which it is interacting.

A.3.1 Agent State

sigaddr-agent (**sigaddr**) the signalling address of the local Agent.

sigaddr-agent ::= sigadddr

maddr-info-agent (**timed-appdata**) the measurement address of the local Agent.

maddr-info-agent ::= timed-appdata

daddr-info-agent (**timed-appdata**) the data address of the local Agent.

daddr-info-agent ::= timed-appdata

root-address (**sigaddr**) the signalling address of the root.

root-address ::= sigaddr

peers (set of **peer-state**) state of each peer.

peers ::= *peer-state

siblings-pending (set of ref of **peer-state**) peers who are potential parents, and have no recent measurements

siblings-pending ::= *peer-state

siblings-measured (set of ref of **peer-state**) peers who are potential parents, and have recent measurements.

siblings-measured ::= *peer-state

current-parent (ref of **peer-state**) the state of the current parent.

current-parent ::= peer-state

tentative-parent (ref of **peer-state**) the peer which this node views as a prospective parent.

tentative-parent ::= peer-state

children (list of ref of **peer-state**, ordered by remote-dist) peers registered as children.

children ::= *peer-state

radius (**appdata**)a radius information given by another peer to help the Agent to find the suitable parent.

radius ::= 1*1appdata

agent-state ::= sigaddr-agent maddr-info-agent daddr-info-agent root-address peers siblings-pending siblings-measured current-parent tentative-parent children radius

A.3.2 Peer State

The Agent holds state about each remote peer with which it interacts.

sigaddr-peer (**sigaddr**) the signalling address of the peer.

sigaddr-peer ::= sigaddr

mode (`ERROR`, `BORING`, `MEASURING`, `TENTATIVE?`, `DISCOVERING`) the state of the current peer in relation to the remote peer.

mode::= Error_mode | Boring_mode | Measuring_mode | Tentative_mode | Discovering_mode

Error_mode ::= '0' *;when an error occurs*

Boring_mode ::= '1' *; the default mode.*

Measuring_mode ::= '2' *; the Agent performs a measurement to this peer.*

Tentative_mode ::= '3' *; the Agent acts as a tentative peer to this peer.*

Discovering_mode ::= '4' *; the Agent performs a Join Procedure with this peer as potential parent.*

maddr-info-peer (**timed-appdata**) the measurement address of the peer.

maddr-info-peer ::= timed-appdata

daddr-info-peer (**timed-appdata**) the data address of the peer.

daddr-info-peer ::= timed-appdata

local-dist (**appdata**) the latest measurement made by this Agent against the peer.

local-dist ::= appdata

remote-dist (**appdata**) the latest measurement made by the peer against this Agent.

remote-dist ::= appdata

leafonly (**boolean**) whether the peer will accept children.

leafonly ::= boolean

parent-peer (ref of **peer-state**) the peer who is thought to be the parent of this peer.

parent-peer ::= peer-state

timeout (**integer**) a period of time (in seconds) the Agent will wait for a response before retransmitting, used to compute expiration times. The messages concerned are: `OBJREQ`, `JOIN` and `NC`.

timeout ::= time

lives (**integer**) the number of times a message will be transmitted without a response.

lives ::= 1*digit

heartbeat-timer (**integer**) the minimum period of time (in seconds) the Agent wait before sending an `ALIVE` message to this peer.

aliveack-wanted (**boolean**) indicates that an ALIVE message was sent to this peer and an `ALIVEACK` message is expected.

join-rsp-pending (**boolean**) indicates a `JOIN` message has been sent, and a `TRY` or `NCA` is expected.

join-rsp-pending ::= boolean

nca-pending (**boolean**) indicates an `NC` message has been sent, and a `NCA` is expected.

nca-pending ::= boolean

daddr-wanted (**boolean**) indicates an `OBJREQ` message has been sent requesting `DADDR`, and a corresponding `OBJRSP` is expected.

daddr-wanted ::= boolean

maddr-wanted (**boolean**) indicates an `OBJREQ` message has been sent requesting `MADDR`, and a corresponding `OBJRSP` is expected.

maddr-wanted ::= boolean

peer-state ::= sigaddr-peer mode maddr-info-peer daddr-info-peer local-dist remote-dist leafonly parent-peer timeout lives heartbeat-timer aliveack-wanted join-rsp-pending nca-pending daddr-wanted maddr-wanted

A.4 Behavior

This section describes the actions (fields update, messages to be sent) to be taken on certain events, most of which are the receipts of messages. The initial state is also described.

A.4.1 Initial State

- Create the User component.
  - Determine the application-specific data address.
  - Decide the type of network stack (e.g. if the Agent is dual stack - IPv4 and IPv6 capable - or single stack).
  - Create the Signalling Address of this node.
- Create the Measurer Component.
  - Determine the application-specific measurement address.
- Create the Agent component.
  - Create this Agent State.

- Create The Transfer component.
  - Create the server(s) according to the type of network stack.
  - Start the server(s).
- Wait for an event from the User component (i.e. *accept*, *join*).

A.4.2 Action On *join* (From User)

- Ignore if already joined to the *root*.
- Clear all state, as if by *leave*.
- Create Peer State for the root. Set root mode to `MEASURING` and maddr-wanted to *true*.
- Set tentative-parent to root
- Send an `OBJREQ` message requesting the measurement address to the root.

A.4.3 Action On *leave* (From User)

- Inform the User that there are no interesting peers.
- Send a `LEAVE` message to all peers.
- Discard all peer states.
- Close all sockets.
- Close the Transfer component

A.4.4 Action On *interest* (From User)

- Inform the User with the n best peers.

A.4.5 Action On *measured* (From Measurer)

The Measurer has obtained a measurement asked for by the Agent.

- Record the measurement in the peer's local-dist.
- If the peer belongs to siblings-pending, move it to siblings-measured. If siblings-pending is now empty, consider sending a `NC` to the tentative-parent, or selecting one of siblings-measured as the new tentative-parent. Break.
- If the peer is the tentative-parent, send a `JOIN`, including the new measurement. Also send a `TENTATIVE` if there is a current-parent and current-parent $\neq$ tentative-parent.

A.4.6 Action On Some Timeouts

A.4.6.1 Action On The Agent Data Address Timeout

- The Agent asks the User for the data address of the node
- Update the Agent state with the new data-address
- Restart timer

A.4.6.2 Action On The Agent Measurement Address Timeout

- The Agent asks the Measurer for the measurement address of the node
- Update the Agent state with the new measurement address
- Restart timer

A.4.6.3 Action On A Peer Data Address Timeout

- Send an `OBJREQ` message requesting `DADDR`.
- Set daddr-wanted to "true".
- Start timer for message reception.

A.4.6.4 Action On A Peer Measurement Address Timeout

- Send an `OBJREQ` message requesting `MADDR`.
- Set maddr-wanted to "true".
- Start timer for message reception.

A.4.6.5 Action On A Message Reception Timeout

- If all flags (maddr-wanted, daddr-wanted. nca-pending and join-rsp-pending) are clear, break.
- If lives > 0
  - For each flag that is not clear, send the message to the node.
  - Deduct one to Lives.
  - Restart timer.

A.4.6.6 Action On A Measurement Timeout

- Ask a measurement to the Measurer.

A.4.6.7 Action On A Maintenance Timeout

- If the Agent is the root of the tree, break
- Go to the *maintenance* state. This involves that the Agent is going to ignore all the ALC messages he will receive, except the `OBJREQ`, `OBJRSP`, `ALIVE` and `ALIVEACK` messages.
- Compute the place of the Agent in the hierarchy
- Choose the ancestor
- Ensure that the measurement is still valid. If it is no more valid, ask the Measurer to perform a *measure*
- If the ancestor is a potential parent closer than the Agent's current one
  - Send a `JOIN` message to this peer.
  - Go to the *Wait* state.
- Else
  - Restart the timer.
  - Go to the *Connected* state.

A.4.6.8 Action On An Error Timeout

- Set the peer's mode to MEASURING.
- Set the state to Init.
- Send an `OBJREQ` message to the tentative-parent.

A.4.6.9 Action On An Heartbeat Timeout

- Send an `ALIVE` message.
- Set aliveack-wanted to "true".
- Start timer for `ALIVEACK` message reception.

A.4.6.10 Action On An `ALIVEACK` Message Reception Timeout

- If the flag aliveack-wanted is clear, break.
- If the peer is one of this peer's children, consider it as dead (i.e. discard its peer state).
- If the peer is the current-parent, proceed like this:
  - If tentative-parent $\neq$ null, break.
  - Set tentative-parent to the parent of the current-parent.
  - Discard current-parent.
  - Ensure tentative-parent mode is measuring.

- If tentative-parent.maddr-info-peer is out of date, send OBJREQ. Break.
- If tentative-parent.local-dist is out of date, send *measure*. Break.
- Send JOIN.

A.4.7 Action On Receipt Of An OBJREQ Message

- Return an OBJRSP. If the *keys* specified DADDR, insert a DADDR containing the Agent daddr-info. If the *keys* specified MADDR, insert a MADDR containing the Agent maddr-info.

A.4.8 Action On Receipt Of An OBJRSP Message

- If a MADDR is present in the *objects*, update the peer maddr-info-peer entry for the sender with its contents. Clear maddr-wanted. If mode is MEASURING, request the measuring component to perform a measurement.
- If a DADDR is present in the *objects*, update the peer daddr-info-peer entry for the sender with its contents. Clear daddr-wanted. If the peer is **[one of the best]**, and the User has not been informed, pass the daddr-info onto the User.
- If all the flags daddr-wanted, maddr-wanted,join-rsp-pending, nca-pending are clear, reset timeout and lives.

A.4.9 Action On Receipt Of A JOIN Message

- If *parent* is absent, and this node is not the root node, respond with an ERROR. Break.
- If *parent* is present, and this node is the root node, or *parent* != current-parent, respond with an ERROR. Break.
- Update the sender's remote-dist from *dist*.
- Update the sender's leafonly state from *leafonly*.
- Identify the set of children (from children) in the same region as the sender.
- If this set is empty, update the sender's *heartbeat-timer* from timer and send a NCA: if a TENTATIVE was present in the JOIN, include one in the response, else record the peer in children. Break.
- If this set is not empty, send a TRY: compute a *radius*, and *sibs* from the set of chosen children, i.e. children falling into the region built by the Agent.

A.4.10 Action On Receipt Of A TRY Message

- If the peer is not the tentative-parent, break.
- Clear join-rsp-pending. If all the flags daddr-wanted, maddr-wanted, join-rsp-pending, nca-pending are clear, reset timeout and lives.
- Record the *sibs* data in the relevant peer states (except for those still in ERROR mode), update siblings-measured and siblings-pending according to whether those peers have an up-to-date measurement and set parent-peer to the sender for each sib.

- Ensure all the peers identified by *sibs* have their mode on measuring.
- For each of *sibs*, send OBJREQ for MADDR.

A.4.11 Action On Receipt Of A NC Message

- If *parent* is absent, and this node is not the root node, respond with an ERROR. Break.
- If *parent* is present, and this node is the root node, or *parent* != current-parent, respond with an ERROR. Break.
- Ensure the state for the sending peer appears in children, according to the new *dist.*
- Update the sender's leafonly state from *leafonly.*
- Update the sender's heartbeat-timer state from *timer*
- Send a NCA message.

A.4.12 Action On Receipt Of A NCA Message

- Record the DADDR and MADDR components into the peer state if present.
- If the peer is not the tentative-parent, break.
- Update the sender's heartbeat-timer state from *timer*
- If there is a current-parent, and the measurement to tentative-parent is no better than that to the current-parent, set the tentative-parent to current-parent, else leave the current-parent, set current-parent to tentative-parent and inform the User about the new parent.

A.4.13 Action On Receipt Of A LEAVE Message

- If sender = current-parent:
  - If tentative-parent is not null, break.
  - Set tentative-parent to the parent-peer of the sender.
  - Set current-parent to null.
  - Ensure tentative-parent mode is measuring.
  - If tentative-parent.maddr-info-peer is out of date, send OBJREQ. Break.
  - If tentative-parent.local-dist is out of date, send *measure.* Break.
  - Send JOIN.
- Else if sender = tentative-parent:
  - If current-parent $\neq$ null, ensure the timer for the maintenance procedure is set, and break.
  - Set the tentative-parent back to an ancestor (the parent-peer), ensure MEASURING, check maddr-info, check local-dist, send JOIN, as before.

A.4.14 Action On Receipt Of An ERROR Message

- Set the mode of the peer to ERROR.
- If the current state is Wait, start the error timer.

A.4.15 Action On Receipt Of An ALIVE Message

- Send an ALIVEACK message in response

A.4.16 Action On Receipt Of An ALIVEACK Message

- If the message is received within the timer, restart the timer for this peer to send a new ALIVE message
- Else break.

A.4.17 Finite State Machine

This section presents a finite state machine (*FSM*). A FSM shows the dynamic of the messages, e.g. how the messages are exchanged and, above all, *when* they can be exchanged between peers (and not exchange of messages/informations inside a peer). The explanation about actions to do on receipt of a particular message are described above. The FSM here represents the interactions of the Agent with the entire hierarchy.

A.4.17.1 The States

- *Init*: this is the initial state. Only the OBJRSP message can be accepted in this state. This state is the beginning of the first join procedure.
- *Wait*: this an intermediate state. It is between a not connected state (*Init*) and a connected state (*Connected*). This state manages the Join Procedure. Of course, while the Agent is in this state, it can't accept a JOIN message from another peer. If the Agent receives an ERROR essage, it directly goes to the *Init* state. If the peer receives a LEAVE or an ERROR message from its tentative-parent, it directly returns to the *Init* state. The following message should be ignored in this state:
  - JOIN
- *Connected*: the Agent is in this state when it is connected to the tree. In this state, the Agent can accept all the messages except the NCA message. If the LEAVE message received comes from a child, there is nothing special to do. Although the LEAVE message comes from the tentative- parent and there is a current-parent, the Agent passes in *Maintenance* state to start a re-join procedure (maintenance procedure as described in section 4.5). If the sender is the current-parent, ensure the "grand-parent" becomes the tentative-parent.
- *Maintenance*: in this state, the Agent perform a re-join procedure (maintenance) as described above (see section 4.5).
- *Finish*: this is the final state, i.e. the state after leaving the tree. In this state, the Agent cannot accept any messages.

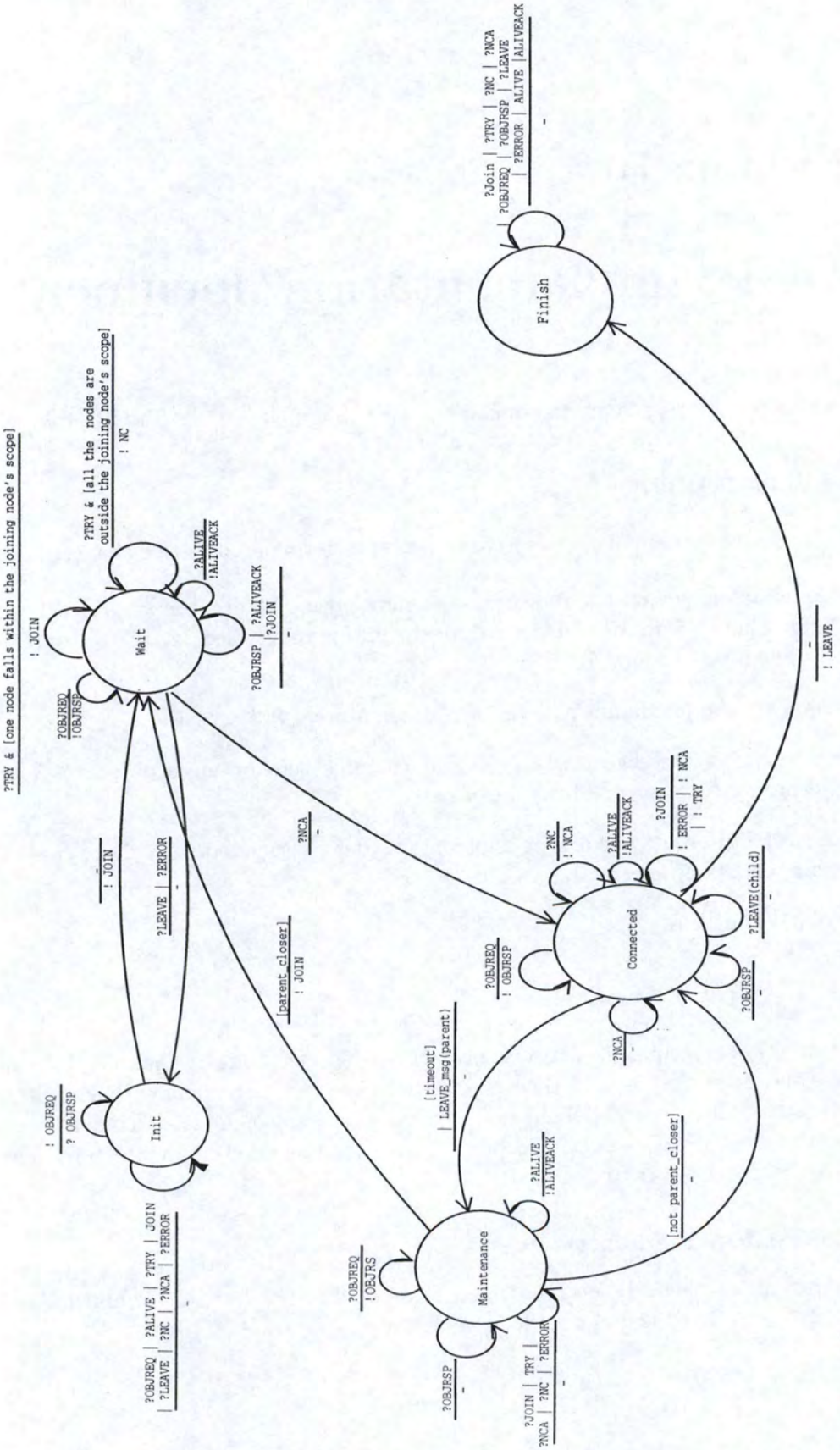


Figure A.11: The Finite State Machine

Appendix B

TBCP: implementation document

This appendix represents the document used to implement the TBCP protocol.

B.1 Terminology

This section contains the definition of concepts/terms used in this chapter.

- *Rendez-vous point*: the root of the spanning tree. The root is identified by the (S, SP) pair, where S is its IP address and SP the port number used. That's the only information a node needs to join the tree.
- *Fanout*: the maximum number of children a node will accept.
- *Message*: a message has a type and contains zero or more objects. A node reacts according to the type of the message.
- *Object*: an object has a type and contains data associated with that type. Objects are components of messages.
- *Node*: end-host.

B.2 Protocol

This section describes the network interactions between nodes in the tree. This includes the message and object types, and their formats. It does not describe how they are transported.

Internal architecture and behavior of a node is implementation-defined, and otherwise beyond the scope of this section. Section 5.1 suggested an internal architecture. The behavior for a peer is described in appendix B.4.

B.2.1 Lexical Elements

We define here lexical elements that will be used in the rest of this appendix. The Augmented BNF (ABNF) [44] is used for the syntax specification.

bit ::= '0' | '1'

Octet ::= 8*8bit

time ::= 2*2Octet

boolean ::= "true" | "false"

digit ::= %30 ... %39 *; 0 ... 9*

objlength ::= 2*2Octet *; object length is stored on 2 bytes*

msglength ::= 4*4Octet *; message length is stored on 4 bytes*

Port ::= 2*2Octet

Address ::= 16*16Octet

B.2.2 Basic Types

Here are some more complex types. They use the lexical elements define in the previous section.

sigaddr ::= 1*2Address *; a signalling address - an IPv6 address and port number and/or, an IPv4 mapped IPv6 address and a port number. It identifies a node in the tree and is sufficient for signalling with that node.*

appdata ::= *Octet *; a sequence of octet with application specific meaning.*

timed-appdata ::= appdata time boolean *; appdata plus an expiry timer and a boolean indicating if the data can be passed to another node.*

positive_integer ::= 1*digit

negative_integer ::= "-"1*digit

integer ::= positive_integer | negative_integer

measurement an application-specific measurement

B.2.3 Object Types

Objects are component of messages. The object type specifies the type of the data it contains and, clearly indicates the purpose of the object. Objects with unrecognized types should be ignored.

B.2.3.1 DADDR Object

This object consists of a **timed-appdata**, and represents the data address of a node.

DADDR_body ::= 1*1timed-appdata

B.2.3.2 MADDR Object

This object consists of a **timed-appdata**, and represents the measurement address of a node.

MADDR_body ::= 1*1timed-appdata

B.2.3.3 MEASUREMENT Object

This object consists of an **appdata**, and represents a measurement.

MEASUREMENT_body ::= 1*1appdata

B.2.3.4 TIMER Object

This object consists of a **timer**, and represents a minimum timer value for the heartbeat message.

TIMER_body ::= 1*1timer

B.2.3.5 NODEADDR Object

This object consists of an **Address**, and represents the signalling address of a node. A second **Address** holds an alternative address for nodes that are dual stack (i.e. IPv4/v6 capable).

NODEADDR_body ::= 1*2Address

B.2.3.6 KEY Object

This object consists of an object identifier (see section B.2.5), and indicates an object being requested.

KEY_body ::= 1*1Obj_id

B.2.3.7 ROOT Object

This object consists of an **Address**, and represents the signalling address of the root. A second **Address** holds an alternative address for nodes that are dual stack (i.e. IPv4/v6 capable).

ROOT_body := 1*2Address

B.2.4 Message Types

Messages with unrecognized type or received at an unexpected moment will be ignored.

B.2.4.1 OBJREQ Message

This message is sent to a node to request objects (for example DADDR, MADDR, ...), and contains the following objects:

keys : a sequence of KEYs

OBJREQ_body ::= *KEY_obj

B.2.4.2 OBJRSP Message

This message is sent as a response to an OBJREQ message and contains the following objects:

Objects : a sequence of objects

OBJRSP_body ::= *Object *; Object will be described in section B.2.5*

B.2.4.3 REJECT Message

This message contains no objects, and is sent by a potential parent to a new comer to tell it that it has to restart the Join Procedure from the beginning.

B.2.4.4 HELLO Message

This message marks the start of a Join Procedure. It is sent by a new comer to a potential parent, and contains the following objects:

root : a ROOT indicating the root of the tree.

parent : an optional NODEADDR indicating the expected parent of the receiver.

HELLO_body ::= 1*2Object

B.2.4.5 HELLOACK Message

This message is sent as a response to an HELLO message and contains the following objects:

maddr : a MADDR indicating the measurement address of the sender

sibs : a sequence of <NODEADDR;MADDR>. Each MADDR is associated with the preceding NODEADDR (the signalling address of a sender's child).

HELLOACK_body ::= 1*Object

B.2.4.6 JOIN Message

This message is sent by a new comer to its potential parent and contains the following objects:

dist : a MEASUREMENT taken by the new comer against its potential parent.

distlist : a sequence of <NODEADDR;MEASUREMENT>. Each MEASUREMENT is associated with the preceding NODEADDR (measures taken by the new comer against the potential parent's children).

JOIN_body ::= 1*Object

B.2.4.7 WELCOME Message

This message is sent by a parent to accept a new comer as its child, and contains the following objects:

maddr : a MADDR indicating the measurement address of the sender

daddr : a DADDR indicating the data address of the sender

timer : a TIMER indicating the minimum timer value for the ALIVE message sent from the new comer to the parent

WELCOME_body ::= 3*3Object

B.2.4.8 WELCOMEACK Message

This message is sent by a new comer to its new parent to acknowledge a WELCOME message, and contains the following objects:

maddr : a MADDR indicating the measurement address of the sender

daddr : a DADDR indicating the data address of the sender

timer : a TIMER indicating the minimum timer value for the ALIVE message sent from the parent to the new comer

WELCOMEACK_body ::= 3*3Object

B.2.4.9 GO Message

This message is sent by a parent to a new comer or one of its children to redirect it to another child, and contains the following objects:

parent : a NODEADDR indicating the signalling address of the new potential parent

GO_body ::= 1*1Object

B.2.4.10 GOACK Message

This message contains no objects, and is sent to acknowledge a GO message.

B.2.4.11 ERROR Message

This message contains no objects, and is sent to express that an error occurs (for example, the node contacted for a Join Procedure is not the potential parent or belongs to another tree).

B.2.4.12 LEAVE message

This message contains no objects, and is sent by a node which wants to leave the tree.

B.2.4.13 ALIVE Message

This message contains no objects, and is sent to a node to test if it is still alive.

B.2.4.14 ALIVEACK Message

This message contains no objects, and is sent to acknowledge an ALIVE message (and, broadly to confirm that the node is still alive).

B.2.5 Object Representation

Each object is transmitted as a sequence of octets, beginning with a length (the maximum length is 2^{16}), an Object-type identifier and a body [1] (the length accounts for all three).

Object ::= objlength Obj_id *Octet

The identifiers for each object are listed below:

Obj_id ::= DADDR_id | MADDR_id | MEASUREMENT_id | TIMER_id | NODEADDR_id | KEY_id | ROOT_id

DADDR_id ::= 0 ; *identifier of the DADDR object*

MADDR_id ::= 1 ; *identifier of the MADDR object*

MEASUREMENT_id ::= 2 ; *identifier of the MEASUREMENT object*

TIMER_id ::= 3 ; *identifier of the TIMER object*

NODEADDR_id ::= 4 ; *identifier of the NODEADDR object*

KEY_id ::= 5 ; *identifier of the KEY object*

ROOT_id ::= 6 ; *identifier of the ROOT object*

An object with a particular type should have a body, as defined in section B.2.3

DADDR_obj ::= objlength DADDR_id DADDR_body

MADDR_obj ::= objlength MADDR_id MADDR_body

MEASUREMENT_obj ::= objlength MEASUREMENT_id MEASUREMENT_body

TIMER_obj ::= objlength TIMER_id TIMER_body

NODEADDR_obj ::= objlength NODEADDR_id NODEADDR_body

KEY_obj ::= objlength KEY_id KEY_body

ROOT_obj ::= objlength ROOT_id ROOT_body

Object ::= DADDR_obj | MADDR_obj | MEASUREMENT_obj | TIMER_obj | NODEADDR_obj | KEY_obj | ROOT_obj

[1] a sequence of octets

B.2.6 Message Representation

Each message is transmitted as a sequence of octets, beginning with a length, a message-type identifier and a body [2] (the length accounts for all three parts). This sequence is encoded in big-endian.

Message ::= msglength Msg_id *Object

The identifiers for each message are listed below:

Msg_id ::= OBJREQ_id | OBJRSP_id | REJECT_id | HELLO_id | HELLOACK_id | JOIN_id | WELCOME_id | WELCOMEACK_id | GO_id | GOACK_id | ERROR_id | LEAVE_id | ALIVE_id | ALIVEACK_id

OBJREQ_id ::= 0 ; *identifier of the OBJREQ message*

OBJRSP_id ::= 1 ; *identifier of the OBJRSP message*

REJECT_id ::= 2 ; *identifier of the REJECT message*

HELLO_id ::= 3 ; *identifier of the HELLO message*

HELLOACK_id ::= 4 ; *identifier of the HELLOACK message*

JOIN_id ::= 5 ; *identifier of the JOIN message*

WELCOME_id ::= 6 ; *identifier of the WELCOME message*

WELCOMEACK_id ::= 7 ; *identifier of the WELCOMEACK message*

GO_id :: = 8 ; *identifier of the GO message*

GOACK_id ::= 9 ; *identifier of the GOACK message*

ERROR_id ::= 10 ; *identifier of the ERROR message*

LEAVE_id ::= 11 ; *identifier of the LEAVE message*

ALIVE_id ::= 12 ; *identifier of the ALIVE message*

ALIVEACK_id ::= 13 ; *identifier of the ALIVEACK message*

A message with particular type should have a body as defined in section B.2.4

[2] a sequence of objects

OBJREQ_msg ::= msglength OBJREQ_id OBJREQ_body

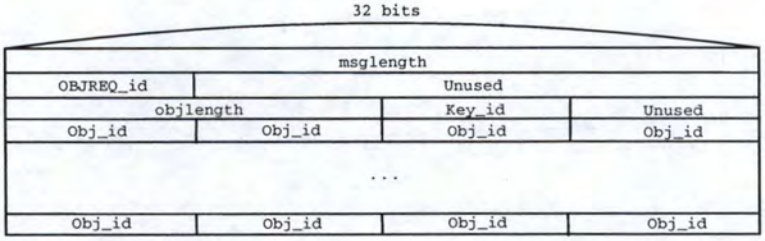


Figure B.1: The OBJREQ message

OBJRSP_msg ::= msglength OBJRSP_id OBJRSP_body

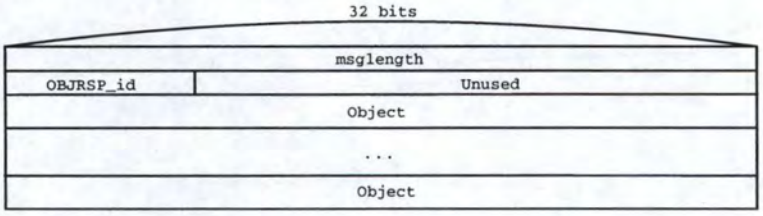


Figure B.2: The OBJRSP message

REJECT_msg ::= msglength REJECT_id REJECT_body

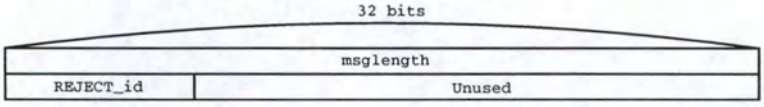


Figure B.3: The REJECT message

HELLO_msg ::= msglength HELLO_id HELLO_body

HELLOACK_msg ::= msglength HELLOACK_id HELLOACK_body

JOIN_msg ::= msglength JOIN_id JOIN_body

WELCOME_msg ::= msglength WELCOME_id WELCOME_body

WELCOMEACK_msg ::= msglength WELCOMEACK_id WELCOMEACK_body

GO_msg ::= msglength GO_id GO_body

GOACK_msg ::= msglength GOACK_id GOACK_body

ERROR_msg ::= msglength ERROR_id

LEAVE_msg ::= msglength LEAVE_id

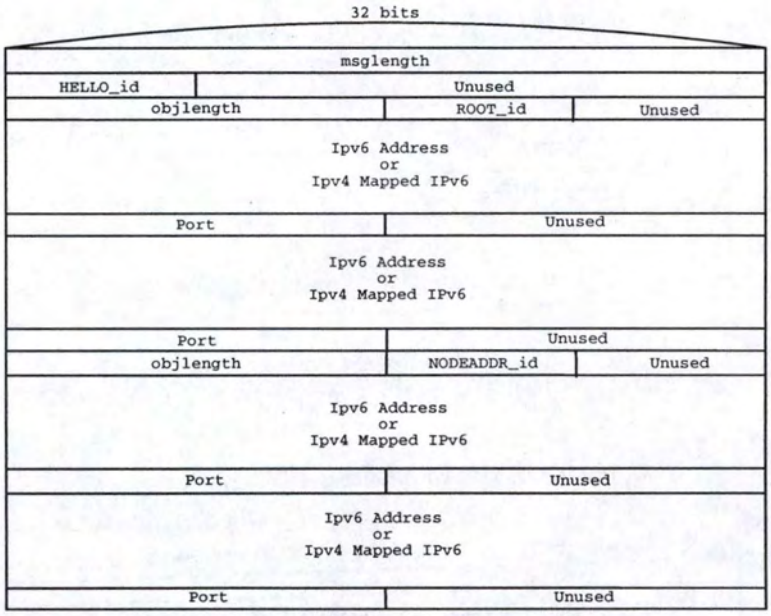


Figure B.4: The HELLO message

ALIVE_msg ::= msglength ALIVE_id

ALIVEACK_msg ::= msglength ALIVEACK_id

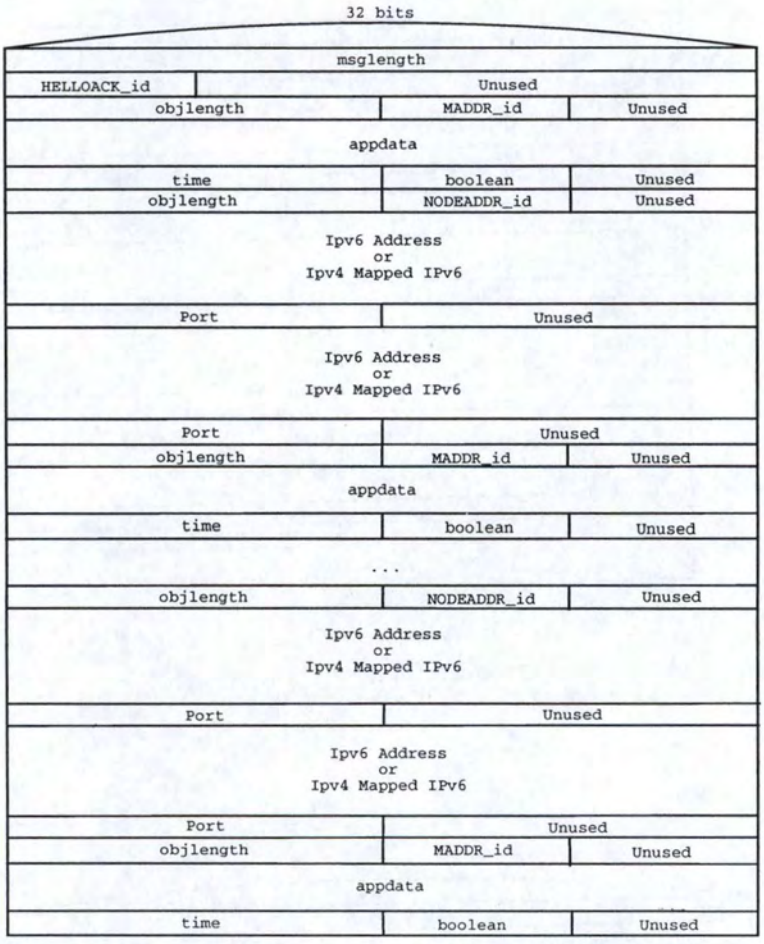


Figure B.5: The HELLOACK message

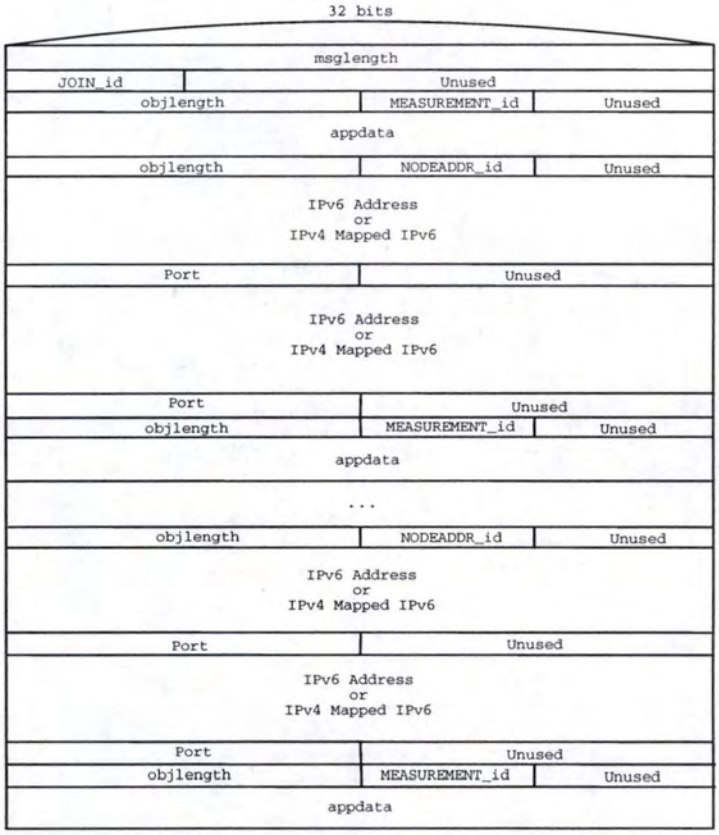


Figure B.6: The `JOIN` message

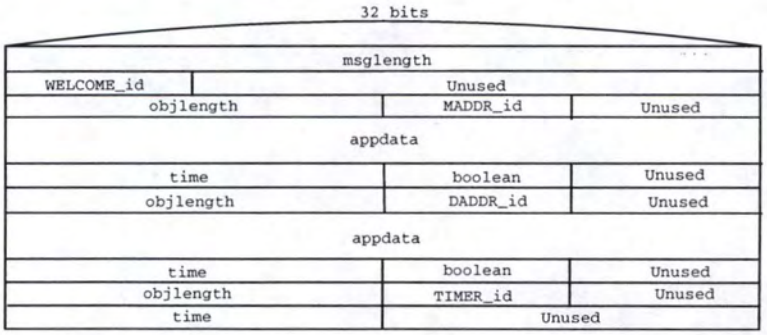


Figure B.7: The `WELCOME` message

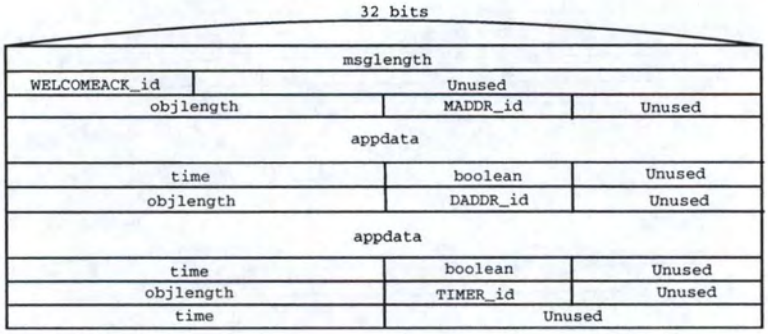


Figure B.8: The WELCOMEACK message

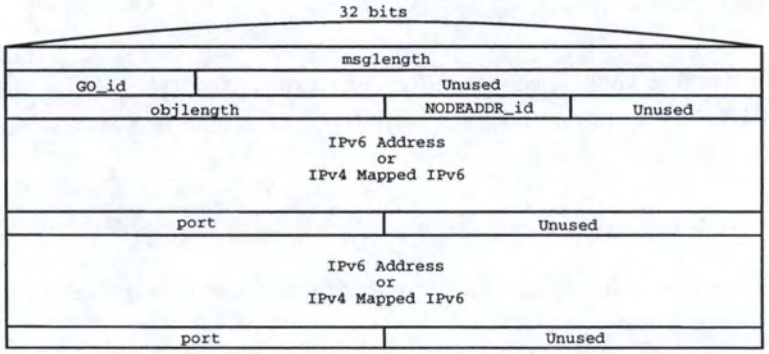


Figure B.9: The GO message

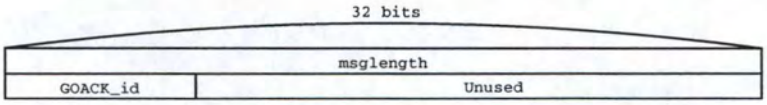


Figure B.10: The GOACK message

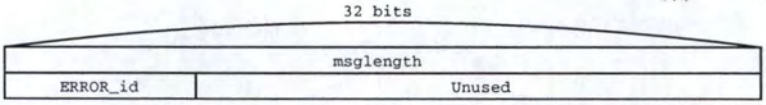


Figure B.11: The ERROR msg

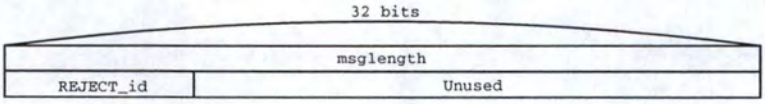


Figure B.12: The LEAVE message

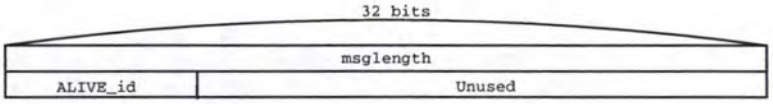


Figure B.13: The ALIVE message

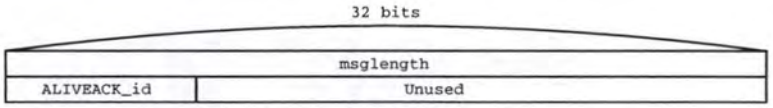


Figure B.14: The ALIVEACK message

B.3 State

A Controller keeps informations about its position in the tree. Some of this (*the Controller state*) applies to its entire participation in the tree, while the rest (*the node state*) applies only to individual nodes with which it is interacting.

B.3.1 Controller State

sigaddr-controller (**sigaddr**) The signalling address of the Controller.

sigadd-controller ::= sigaddr

maddr-controller (**timed-appdata**) The measurement address of the Controller.

maddr-controller ::= timed-appdata

daddr-controller (**timed-appdata**) The data address of the Controller.

daddr-controller ::= timed-appdata

fanout (**integer**) The maximum number of children that will be accepted by the Controller.

fanout ::= positive_integer

root (**sigaddr**) The signalling address of the root of the tree.

root ::= sigaddr

timer-join (**time**) The timer value (in seconds) for the Join Procedure. After this timeout, the Join Procedure must be completed.

timer-join ::= time

nodes (set of **node-state**) The list of all nodes the Controller is in contact with.

nodes ::= *node-state

current-parent (**node-state**) The current parent of the Controller.

current-parent ::= node-state

tentative-parent (**node-state**) The node the Controller views as a potential parent.

tentative-parent ::= node-state

children (list of **node-state** ordered by dist).

children ::= *node-state

tentative-child (**node-state**) The potential new child.

tentative-child ::= node-state

measure-pending (set of **node-state**) Nodes that have no recent measurements.

measure-pending ::= *node-state

measure-performed (set of **node-state**) Nodes that have recent measurements.

measure-performed ::= *node-state

controller-state ::= sigaddr-controller maddr-controller daddr-controller fanout root timer-join nodes current-parent tentative-parent children tentative-child measure-pending measure-performed

B.3.2 Node State

sigaddr-node (**sigaddr**) The signalling address of this node.

sigaddr-node ::= sigaddr

maddr-node (**timed-appdata**) The measurement address of this node.

maddr-node ::= timed-appdata

daddr-node (**timed-appdata**) The data address of this node

daddr-node ::= timed-appdata

parent-node (**node-state**) The node presumed to be the parent of this node.

parent-node ::= node-state

dist (**appdata**) The distance between the Controller and this node.

dist ::= appdata

dist-newcomer (**appdata**) The distance between the new comer and this node.

dist-newcomer ::= appdata

heartbeat-timer (**time**) The minimum period of time (in seconds) the Controller will wait before sending an `ALIVE` message to this node.

heartbeat-timer ::= time

alive-ack-wanted (**boolean**) Indicates that an `ALIVE` message has been sent to this node and a corresponding `ALIVEACK` is expected.

alive-ack-wanted ::= boolean

daddr-wanted (**boolean**) Indicates that an `OBJREQ` message requesting a `DADDR` object has been sent and a corresponding `OBJRSP` is expected.

daddr-wanted ::= boolean

maddr-wanted (**boolean**) Indicates that an `OBJREQ` message requesting a `MADDR` object has been sent and a corresponding `OBJRSP` is expected.

maddr-wanted ::= boolean

join-wanted (**boolean**) Indicates that a `HELLOACK` message has been sent and a `JOIN` message is expected.

join-wanted ::= boolean

lives (**integer**) A number of times a message will be sent without a response.

lives ::= integer

timeout (**time**) a period of time (in seconds) the Controller will wait for a response before retransmitting.

timeout ::= time

mode (`ERROR`, `MEASURING`, `BORING`) The "state" of the node

mode ::= `ERROR_mode` | `MEASURING_mode` | `BORING_mode`

ERROR_mode ::= "0" *; This node sends back an `ERROR` message to the Controller*

MEASURING_mode ::= "1" *; The measurer of the node is performing measurement to this node*

BORING_mode ::= "2" *; when everything is alright, a node is in the BORING mode*

node-state ::= sigaddr-node maddr-node daddr-node dist dist-newcomer heartbeat-timer aliveack-wanted daddr-wanted maddr-wanted join-wanted lives timeout mode

B.4 Behavior

This section describes the actions (fields updates, messages to be sent) to be taken on certain events, most of which are the receipts of message. The initial state is also described.

B.4.1 Initial State

- Create the User component.
- Determine the application-specific data address
- Decide the type of network stack (e.g. if the Controller is dual stack - IPv4 and IPv6 capable - or single stack).
- Create the signalling address of this node.
- Create the Measurer Component.
- Determine the application-specific measurement address.
- Create the Controller Component.
- Create the Controller State
- Create The Transfer Component
- Wait for an event from the User component (i.e. *accept*, *join*)

B.4.2 Action On *join* (From The User)

- Ignore if the node has already joined the tree.
- Create node state for the root.
- Set root as tentative-parent.
- Send an `HELLO` message.
- Go to the *Wait* state.

B.4.3 Action On *accept* (From The User)

- Go to the *Connected* state.

B.4.4 Action On *leave* (From The User)

- Send a `LEAVE` message to all nodes.
- Discard all node state.
- Close all sockets.
- Close the Transfer component.

B.4.5 Action On *measured* (From The Measurer)

The Measurer has obtained a measurement asked by the Controller.

- Record the measurement in the node's dist.
- If the node belongs to measure-pending, move it to measure-performed
- If the measure-pending is now empty, send a `JOIN` message to the tentative-parent containing these distances.

B.4.6 Action On Some Timeouts

B.4.6.1 Action On The Controller Data Address Timeout

- The Controller asks the User for the data address of the node (event *getAddress()⇒daddr*).
- Update the Controller's daddr-controller with *daddr*.
- Restart timer.

B.4.6.2 Action On The Controller Measurement Address Timeout

- The Controller asks the Measurer for the measurement address of the node (event *getAddress()⇒ maddr*).
- Update the Controller's maddr-controller with *maddr*.
- Restart timer.

B.4.6.3 Action On A Node's Data Address Timeout

- Send an `OBJREQ` message requesting `DADDR`.
- Set daddr-wanted to "true".
- Start timer for message reception.

B.4.6.4 Action On A Node's Measurement Address Timeout

- Send an `OBJRSP` message requesting `MADDR`.
- Set maddr-wanted to "true".
- Start timer for message reception.

B.4.6.5 Action On A Message Reception Timeout

- If all flags (maddr-wanted and daddr-wanted) are clear, break.
- If lives > 0
  - For each flag that is not clear, send the associated message to the node.
  - Deduct one to Lives.
  - Restart timer.

B.4.6.6 Action On A Maintenance Timeout

- If the node is the root of the tree, break.
- Go to the *Maintenance* state.
- Compute the place of the node in the tree.
- Choose the ancestor.
- If the ancestor chosen is the current-parent, break.
- Set tentative-parent to the chosen ancestor.
- Go to the *Wait* state.
- Start a Join Procedure and use the tentative-parent as rendez-vous point.

B.4.6.7 Action On A Heartbeat Timeout

- Send an `ALIVE` message.
- Set aliveack-wanted to "true".
- Start timer for `ALIVEACK` message reception.

B.4.6.8 Action On An `ALIVEACK` Message Reception Timeout

- If the flag aliveack-wanted is clear, break.
- If the node is one of this node's children, consider it as death (i.e. discard its node state).
- If the node is the current-parent, proceed like this:
  - If tentative-parent $\neq$ null, break.
  - Set tentative-parent to the parent of the current-parent.
  - Discard current-parent.
  - Go to the *Wait* state.
  - Start a Join Procedure and use the tentative-parent as rendez-vous point.

B.4.6.9 Action On A Join Procedure Timeout

- If tentative-child's join-wanted is clear, break.
- Else
  - Send a `REJECT` message to the tentative-child.
  - Release all node state associated.
  - Go to the *Connected* state.

B.4.6.10 Action On An Error Timeout

- Send an HELLO message to the tentative-parent.
- Go to the *Wait* state.

B.4.7 Action On Receipt Of An OBJREQ Message

Return an OBJRSP message.

- If *keys* specified a DADDR, insert a DADDR containing the Controller daddr-controller.
- If *keys* specified a MADDR, insert a MADDR containing the Controller maddr-controller.

B.4.8 Action On Receipt Of An OBJRSP Message

- If a MADDR is present in the objects, update the maddr-node information for the sender, and clear maddr-wanted. If mode for the sender is MEASURING_mode, ask the Measurer to perform a measurement.
- If a DADDR is present in the objects, update the daddr-node information for the sender, and clear daddr-wanted.
- If all flags (madr-wanted, daddr-wanted) are clear, reset timeout and lives.

B.4.9 Action On Receipt Of A REJECT Message

- If the sender is not the tentative-parent, break.
- Clear the sender's node-state.
- Wait during a random time.
- Restart a Join Procedure (i.e. send an HELLO message to the tentative-parent.

B.4.10 Action On Receipt Of A HELLO Message

- If *parent* is absent and this node is not the root node, send an ERROR message. Break.
- If *parent* is present and this node is the root node, or *parent* != current-parent, send an ERRROR message. Break.
- Create a node-state for the sender.
- Record the sender as tentative-child.
- Set tentative-child's join-wanted to true.
- Send an HELLOACK message containing the maddr-controller information and a "siblist" (i.e. a list of the signalling address and the measurement address of each child).
- Start the timer for the Join Procedure.
- Go to the *JoinProc* state.

B.4.11 Action On Receipt Of A HELLOACK Message

- If the sender is not the tentative-parent, break.
- Update tentative-parent with MADDR information.
- Ensure the tentative-parent's mode is MEASURING.
- Add tentative-parent to measure-pending.
- Create a node-state for each <NODEADDR, MADDR>.
- Ensure the mode is MEASURING for each pair.
- Add each pair to measure-pending and to nodes.
- For each node in the measure-pending list, send an OBJREQ message for MADDR.

B.4.12 Action On Receipt Of A JOIN Message

- If the sender is not the tentative-child, break.
- Clear join-wanted.
- If | children | + 1 $\leq$ fanout
  - Send a WELCOME message to the tentative-child including MADDR, DADDR and TIMER.
  - Break.
- If | children | + 1 > fanout
  - Compute the score function based on the measurements included in the JOIN message.
  - Choose the best local configuration.
  - If the tentative-child has to be redirected.
    - Send a GO message including the new rendez-vous point to the tentative-child.
    - Remove tentative-child.
  - If a node's child must be redirected
    - Send to it a GO message including its rendez-vous point.
    - Remove its node state.
    - Send the tentative-child a WELCOME message including MADDR, DADDR, Timer information.

B.4.13 Action On Receipt Of A WELCOME Message

- If the sender is not the tentative-parent, break.
- Record the MADDR, DADDR and TIMER informations in the relevant node state.
- Move tentative-parent to current-parent.

- Send back a WELCOMEACK message including DADDR, MADDR and TIMER informations.
- Start the timer for the heartbeat message.
- Start the timer for the Maintenance Procedure.
- Go to the *Connected* state.

B.4.14 Action On Receipt Of A WELCOMEACK Message

- If the sender is not the tentative-child, break.
- Record the MADDR, DADDR and TIMER informations in the relevant node state.
- Add tentative-child to children.
- Start the timer for the heartbeat message.
- Go to the *Connected* state.

B.4.15 Action On Receipt Of A GO Message

- If the sender is not the current-parent or the sender is not the tentative-parent, break
- Create a node state for the NODEADDR included in the GO message
- Set node-parent to sender.
- Record the node state as tentative-parent
- Send back a GOACK message.
- Start a Join Procedure and use the tentative-parent as the rendez-vous point.

B.4.16 Action On Receipt Of A GOACK Message

- If the sender is the tentative-child, release all node states related and go to the *Connected* state.
- If the sender is a node's child, release all node states related and go to the *Connected* state.

B.4.17 Action On Receipt Of An ERROR Message

- If the sender is not the tentative-parent, break.
- Set the sender's mode to ERROR.
- Start the timer for error.
- Go to the Init state.

B.4.18 Action On Receipt Of An ALIVE Message

- If the sender is unknown, break.
- Send back an ALIVEACK message.

B.4.19 Action On Receipt Of An ALIVEACK Message

- If the sender is unknown, break.
- If the message was received within the timer, clear aliveack-wanted and restart timer.
- Else, break.

B.4.20 Action On Receipt Of A LEAVE Message

- If the sender is a node's child, release all node states related. Break.
- If sender = current-parent
  - If tentative-parent is not null, break.
  - Set tentative-parent to the parent-node of the sender.
  - Release current-parent
  - Start a Join Procedure and use the tentative-parent as rendez-vous point.
  - Go to the *Wait* state.
- If sender = tentative-parent
  - If current-parent $\neq$ null, ensure the timer for the maintenance procedure is set and break.
  - Set tentative-parent back to an ancestor (parent-node), start a Join Procedure, use the tentative-parent as rendez-vous point and go to the *Wait* state.

B.4.21 Finite State Machine

This section presents the Finite State Machine (*FSM*). The FSM represents the interactions of the Controller with the entire tree.

B.4.21.1 The States

- *Init*: This is the initial state. None message can be accepted in this state. This state is the beginning of the Join Procedure.
- *Wait*: This is an intermediate state. It is between a not connected state (*Init*) and a connected state (*Connected*). In this state, a new comer wait for a WELCOME message or a GO message. If the node receives a WELCOME message, it goes to the *Connected* state (it is now considered as a child of another node). If it receives a GO message, it returns to the *Init* state and has to restart a Join Procedure.

- *Connected*: In this state, the node is connected to the tree. If the node is the root node, it directly goes to this state. In this state, a node can receive a `GO` message from its parent. It then goes to the *Init* state to start a Join Procedure. In this state, the following messages should be ignored because they are parts of a Join Procedure:
  - `WELCOME` message.
  - `WELCOMEACK` message.
  - `JOIN` message.
  - `HELLOACK` message.
  - `REJECT` message.
  - `GOACK` message.
  - `ERROR` message.

  The node can leave the tree by sending a `LEAVE` message. It then goes to the *Finish* state when it receives a *leave* event from the User. However, if the node receives an `HELLO` message, it answers with an `HELLOACK` message and goes to the *JoinProc* state. Now, the node plays the role of a rendez-vous point.

- *JoinProc*: When a node passes into this state, it starts a timer for the Join Procedure. If the `JOIN` message is not received within this timer, the node sends the potential child a `REJECT` message and returns to the *Connected* state. On the opposite, the node sends back a `WELCOME` or `GO` message and wait for the corresponding acknowledgement. When it is received, the node returns to the *Connected* state. The following messages are not accepted in this state:
  - `HELLO` message (a node can only deals with one Join Procedure at a time for consistency reasons).
  - `HELLOACK` message (a node can't be at the same time a new comer and a rendez-vous point).
  - `ERROR` message (a node can only accept this message when it performs a Join Procedure as a new comer).

  During a Join Procedure, the node can leave the tree (if the User decided so) and to do so, sends a `LEAVE` message and goes to the *Finish* state.

- *Maitenance*: In this state, the Controller performs a Maintenance Procedure as described above (see section 4.5). All messages, except the `ALIVE`, `ALIVEACK`, `OBJREQ` and `OBJRSP` messages should be ignored.

- *Finish*: This is the final state, i.e. the state after leaving the tree. In this state, the Controller can't accept any messages.

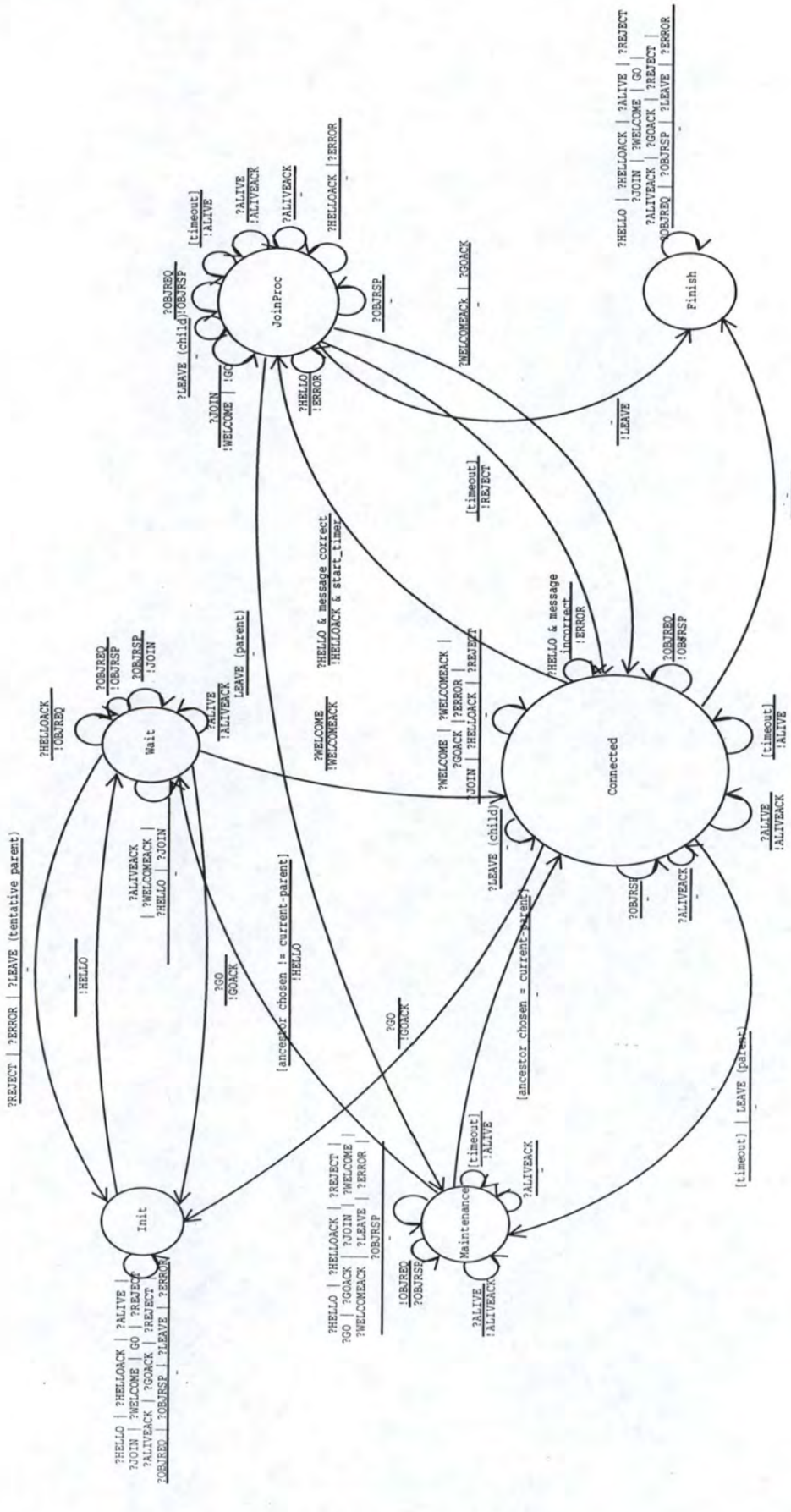


Figure B.15: The Finite State Machine